CRWR Online Report 03-07

Identification of large woody debris in acoustic bathymetry data

by

Laurent White, B.S.
Graduate Research Assistant

and

Ben R. Hodges, Ph.D.
Principal Investigator

The University of Texas at Austin

September 30, 2003

Final Report to Texas Water Development Board
under Contract No. 2003-469

Abstract

The more frequent use of two-dimensional hydrodynamic river models also requires more detailed bathymetry surveys. For smooth bathymetries, there is little difficulty in developing accurate translations from survey data to model; however, in rivers with significant bottom structure (e.g., large woody debris), simple data averaging and interpolation methods may lead to misrepresentation of the bottom bathymetry. It is necessary to identify in the data set what is true bathymetry from what is caused by large woody debris. Two groups of methods are investigated to serve our objective: statistical techniques and filtering techniques. While the former are appealing for their simplicity and direct applicability in modeling, they fail at consistently treating spikes (hypothesized to be large woody debris signature) in the data set. Among filtering techniques, linear filters are turned down due to their inherent trade-off between edge retention and spike rejection. Two nonlinear filters are examined. Median and erosion filters are specifically designed to preserve sharp edges while eradicate spikes. Finally, median filtering is preferred to erosion filtering, for the former leaves large-scale bathymetric features virtually undisturbed.

Contents

List of Figures

Chapter 1

Introduction

As the trees growing alongside a stream or river age, die and decay, large branches and sometimes even the whole trunk can fall or topple onto the streambank or into the channel itself. We commonly refer to this amount of woody material as large woody debris (LWD). The dimensions of LWD are usually taken to be greater than 0.1 m in diameter and 1.0 m in length.

Figure 1.1: Emergent LWD in the Sulphur River (Northeast Texas) at a low flow rate during ?? 20??, when a boat-conducted bathymetric survey would be impractical. Debris is submerged at high flow, when bathymetry surveys are typically performed. (photograph courtesy of Texas Water Development Board).

To the early settlers, LWD was often a nuisance and these fallen trees and branches were usually termed snags. They made access to streams by stock difficult, and large snags within rivers were a major hazard to transport and navigation at a time when waterways were a major route for moving goods and people. This use of rivers involved periodic or regular removal of obstructions as a part of so-called river improvement, river clearing or channelization schemes (Gippel, 1995). In addition to enhancing river navigability, the removal of snags has often been justified on the grounds that it improves water conveyance, reduces bank erosion, rejuvenates channels, lessens the risk of damage to bridges, improves recreational amenity and removes barriers to fish migration (Harmon *et al.*, 1986).

Snag management has generally been regarded as an engineering or economic issue, and because of this narrow focus, most snag removal has been undertaken with little concern for the environmental role of snags and, in particular, their direct or indirect effects on aquatic fauna and flora. It is now well recognized and established that fallen wood in streams has a multifunctional and positive role on an environmental point of view. Several reviews of the literature provide grounds for this assertion (e.g., Shields and Nunnally (1984) and Harmon *et al.* (1986)). Research over the past 20 years has actually shown that woody debris is a vital component for the healthy functioning of rivers. For this reason, it has become far more appropriate to use the term large woody debris instead of snag when referring to fallen wood in streams.

In the past two decades, the large majority of hydraulic studies regarding large woody debris have focused on their stream-scale management. Research in stream restoration (Shields and Nunnally (1984), Shields and Gippel (1995), Gerhard and Reich (2000), Gippel *et al.* (1996a)) has generally gravitated around the common issue of achieving desirable hydraulic effects (e.g., decrease flow resistance) while minimizing undesirable environmental effects (e.g., the loss of aquatic habitat diversity). The focus has also been directed toward the ecological and morphological effects of large woody debris but very little has been done regarding the local flow pattern around debris such as velocity distribution, turbulence intensities and secondary currents. As noted by Mutz (2000), the local flow pattern is controlled by the woody debris and the former has been well established to be highly significant to fauna and flora, as summarized by Gippel (1995) and determined by Kemp *et al.* (2000).

1.1 Cause-effect relationships of LWD

LWD is known to influence the fluid mechanics, ecology and morphology of streams in many ways. Let us first clarify what these three aspects of a stream mean to us. The *fluid mechanics* encompasses the properties, distribution and circulation of water. The study of secondary currents, standing water or turbulence within a stream belongs to the field of fluid mechanics. The *ecology* is concerned with the pattern of relations between organisms and their environments. Why some invertebrates are more likely to live in regions of standing water is an ecological matter. Finally, the *morphology* deals with the structure and form of the stream. These properties of a stream such as its sinuosity, its order or its pools distribution fall within the scope of the morphological study of the stream.

The diagram in Figure 1.2 shows the *a priori* direct and indirect effects of LWD that one should expect. Despite the previous definitions and the seemingly well-defined relationships in the diagram, we will see that the latter are unfortunately not as clear-cut as one would anticipate.

1.1.1 Effects of LWD on stream ecology

In addition to the many indirect effects of LWD on ecology – e.g., LWD creates hydraulic diversity that enhances fish species diversity –, the presence of LWD has a direct impact on ecology. Macroinvertabrates benefit from the structural complexity provided by debris (Minshall, 1984). Furthermore, large items of debris provide a secure, hard surface upon

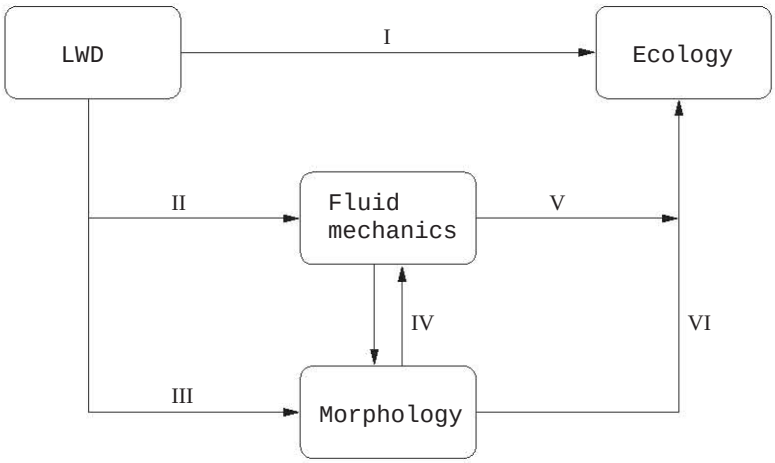

Figure 1.2: Cause-effect relationships of large woody debris within stream environments. An arrow reads *affects*

which microscopic plants (algae) can grow, and provide habitat for aquatic invertebrates such as insect larvae and snails. LWD helps to trap leaf litter and other organic matter moving downstream to form debris dams, which become hot-spots of biological activity and a major source of food for animals (Land and Water Australia, 2002). Animals feeding on algae or involved in shredding and consuming leaves and fine litter are key components of aquatic systems because they, in turn, become food for larger river animals such as crustaceans, fish and platypus. In this way, LWD plays an important role in providing a base for the processing of energy and nutrients to support the aquatic food web.

Large debris is also vital for the survival and growth of many important fish species. It provides habitat and shelter from predators, while hollow logs are an essential spawning habitat for native fish species; for example the Mary River Cod of south-east Queensland (Australia), and the River Blackfish of Victoria and Tasmania (Australia) require submerged hollow logs in which to lay and nurse their eggs.

These are a few examples of the direct influence of LWD on aquatic life diversity and illustrate the ecological importance of woody debris in rivers. However, this specific role of LWD is not central to our study and will not be investigated further, but it was worth including it for the sake of completeness.

1.1.2 Effects of LWD on stream fluid mechanics

Understanding how LWD affects the flow in streams, locally or at the channel scale, is the key topic of our study. Hydraulic diversity created by LWD is beneficial to the stream ecosystem. This same local flow diversity positively affects the stream morphology, which, in turn, is ecologically beneficial. These complex interactions serve as a justification to the central position of the hydrological aspect in the study of LWD in streams.

Flow resistance

Vegetation and debris increase flow resistance (or roughness) that has a direct effect on the discharge capacity and the level of stream flooding hazard (Dudley *et al.*, 1998). It has been established that debris act as large roughness elements that provide a varied flow environment, reduce average velocity and locally produce an increase in water level (afflux) (Gippel, 1995). The latter is caused by the so-called blockage effect of debris and means that for a given discharge, the water level is higher than without the debris, thereby theoretically increasing flooding frequency at locations upstream of the blockage. Depending on how the river and floodplain are managed, this effect may be perceived as positive (e.g., beneficial for wetlands) or negative (e.g., inconvenient for landholders) (Gippel *et al.*, 1996b).

Two widely-used approaches to quantify the resistance that debris offers to flow make use of a flow resistance equation, in which a roughness coefficient (Manning's n) or a friction factor (Darcy-Weisbach friction factor) is employed. Both methods may be regarded as *zero-dimensional* because they do not attempt to locally model the flow but consider the reach as a whole. Furthermore, as pointed out by Gippel (1995), the Manning equation (i.e. the roughness coefficient approach) is not strictly applicable in the case of LWD, for it was developed to describe open-channel situations where friction is controlled by surface drag from the bed sediments, rather than form drag from large obstacles such as debris. Also, the hydraulic radius, as conventionally defined in Manning's equation, is probably meaningless in channels heavily obstructed with debris.

Both approaches require the evaluation of an hydraulically meaningful debris drag coefficient. In practice, LWD are geometrically approximated as circular cylinders for which drag coefficients in flow of infinite extent (no boundary interference) are well defined. However, for real woody debris, two deviations from this idealistic situation are generally encountered:

- *Debris irregularities.* Woody debris are rarely perfectly circular cylinders. Branches and leaves may significantly increase the drag force and underestimation of the latter may occur if these irregularities exist and are neglected. Gippel *et al.* (1996a) measured the drag coefficient of tree-shaped models compared with that of a cylinder. Four stages of assembly were considered: trunk only (with three short, projecting elbow joiners); trunk and butt; trunk and branches; and complete with trunk, branches and butt. A lower overall drag coefficient for the complete tree-shaped debris model compared with that of a cylinder was obtained and can be explained by the fact that the drag coefficient was expressed relative to the projected surface area. Unlike the simple cylinder model, some flow could pass through the branching section thereby increasing the total drag force but the drag coefficient was lowered because the increase in projected surface area was proportionally greater than the increased drag force. The results of these measurements permitted Gippel *et al.* (1996a) to establish best-fit empirical expressions for the drag coefficient of different debris models as functions of debris orientation.

- *Effect of confined flow.* The blockage effect of confined flow does not alter the inherent drag coefficient C_d of a cylinder. Rather, C_d measured in confined flow is an apparent drag coefficient. In (Ranga Raju *et al.*, 1983), the drag coefficient for a vertical cylinder

of diameter d in a flume of width w is given by an equation of the form

$$C_d = \frac{C'_d}{a\left[1 - \frac{d}{w}\right]^b}, \tag{1.1}$$

where C'_d is the drag coefficient in a flow of infinite extent (no boundary effects) and a and b are determined experimentally. Although debris formations are not vertical cylinders, a series of flume tests on model debris by Shields and Gippel (1995) verified the form of this equation for debris formations, and provided values for coefficients a and b. The ratio $\frac{d}{w}$ was substituted by the blockage ratio B:

$$B = \frac{Ld}{A}, \tag{1.2}$$

where L is the projected length of debris in flow, d is the diameter of debris in flow and A is the cross-sectional area of flow.

The Manning's equation The Manning equation for mean flow velocity, V, reads

$$V = \frac{1}{n} R^{2/3} \sqrt{S_e}, \tag{1.3}$$

where R is the hydraulic radius, S_e is the slope of the energy grade line and n is Manning roughness coefficient. The utilization of this equation implies that all resistive effects – such as vegetation, debris and other obstructions, bed roughness, channel meandering and streambank irregularity – are lumped together and accounted for by a single coefficient.

Dudley *et al.* (1998) studied the effect of woody debris on Manning roughness coefficient by using the relation for Manning's n presented by Petryk and Bosmajian (1975) in Dudley *et al.* (1998). A balance between drag and gravitational forces leads to

$$n = R^{2/3} \left[\frac{C_d Veg_d}{2g}\right]^{1/2} \tag{1.4}$$

where C_d is the drag coefficient of vegetation, Veg_d is the vegetation density and g is the gravitational acceleration. The vegetation density is defined by

$$Veg_d = \frac{\sum A_v}{aR}, \tag{1.5}$$

where $\sum A_v$ is the frontal area of the vegetation projected onto a plane perpendicular to the direction of flow and a is a unit surface area of the channel bed. The development of Eq. (1.4) is reproduced in details in Dudley *et al.* (1998). Measurements prior to and following the removal of woody debris indicated that the average Manning's n value was 39 percent greater when woody debris was present. It was also observed that the impact of

debris on the value of Manning's n decreased with an increase in unit discharge, suggesting a convergence of channel roughness of cleared and uncleared reaches at high flows. We may therefore expect Manning's n value to be constant at high flow rates and woody debris to have little impact on total resistance.

The Darcy-Weisbach friction factor A technique for partitioning the total resistance into various components – the resistance due to woody debris being one component – was developed by Shields and Gippel (1995). A different Darcy-Weisbach friction factor is associated with each resistive component, thereby allowing for a more accurate analysis of the effect of the sole woody debris on flow resistance. The method is based on the assumption that the flow around woody debris can be evaluated on reach level and assumed to be uniform. The authors admit that this approach consists in a gross simplification of the complex nonuniform flow that often occurs around and through debris formations.

The following balance is assumed to hold in a uniform flow on a control volume of length L:

$$F_{\mathrm{g}} = F_{\mathrm{bed}} + F_{\mathrm{bends}} + F_{\mathrm{LWD}} \tag{1.6}$$

where F_g is the force of gravity, F_{bed} is the resistance force due to bed (grain and bar resistance), F_{bends} is the resistance force due to bends and F_{LWD} is the drag force on debris.

It can be shown that the above formula may be rewritten as

$$S_0 = \frac{\tau_b}{\gamma R_{av}} + \frac{\sum \left[B_i / r_{c_i}\right] \alpha V_{av}^2}{gL} + \frac{\sum D_i}{\gamma A_{av} L}, \tag{1.7}$$

where S_0 is the average bed slope, τ_b is the shear stress on boundaries, γ is the specific weight of water, R_{av} is the mean hydraulic radius, B_i is the ith bend water surface width, r_{c_i} is the ith bend radius of curvature, α is the kinetic energy correction factor (assumed to be 1.15 in Henderson (1966)), V_{av} is the mean flow speed, $\sum D_i$ is the resistance due to debris and A_{av} is the fluid control volume divided by the reach length L.

The Darcy-Weisbach equation for uniform flow in an open channel is

$$S_0 = \frac{f \alpha V_{av}^2}{8 g R_{av}}, \tag{1.8}$$

where f is the Darcy-Weisbach friction factor representing total flow resistance. Now, the idea is to partition the friction factor into four components:

$$f = f_{\mathrm{grain}} + f_{\mathrm{bedform\ and\ bar}} + f_{\mathrm{bends}} + f_{\mathrm{debris}} \tag{1.9}$$

so that the last term of (1.7) may be expressed as

$$\frac{\sum D_i}{\gamma A_{av} L} = \frac{f_{\text{debris}} \alpha V_{av}^2}{8 g R_{av}}. \tag{1.10}$$

Finally, the form drag of a piece of solid wood in flow is

$$D_i = \frac{C_{d_i} \gamma V_i^2 A_i}{2g} \tag{1.11}$$

where V_i is the upstream (approach) velocity of ith debris formation and A_i is the projected area of ith debris formation. By combining (1.10) and (1.11), we can solve for the Darcy-Weisbach friction factor due to debris, provided that the drag coefficient C_{d_i} be properly assessed.

Field experiments in cleared and uncleared reaches of the Obion and Tumut rivers (Australia) were carried out by Shields and Gippel (1995) in order to

1. evaluate how close the computed value of the Darcy-Weisbach friction factor was relative to the measured value in the field;

2. evaluate the impact of the presence or removal of woody debris on the total friction factor under different flow conditions.

The study site of the Obion River was straight and the bed was mainly made of sand. The Tumut River channel in the study area was a sinuous, fast-flowing river with a bed mainly composed of gravel (75 %). In both rivers, computed values of Darcy-Weisbach friction factor were slightly more accurate for reaches with debris (errors ranged from -28 to +19 %) than for reaches without debris (errors ranged from -38 to +30 %). The mean of the absolute values of errors was lower for the straight sand-bed Obion (13 %) than for the sinuous, gravel-bed Tumut (19 %).

As regards debris removal, modest effects were observed. Increases of 6 % and 22 % in flow conveyance were reported in the Tumut River and Obion River, respectively.

Another study by Manga and Kirchner (2000) centered on the estimation of the partitioning of flow shear stress between woody debris and streambeds. Their measurements showed that, even though LWD covered less than 2 % of the streambed, they provided roughly half of the total flow resistance. This result was obtained by using different methods of measurement. One of these consisted in inferring the drag from water surface steps, using conventional energy balance arguments. It is now well established that woody debris causes a perceptible afflux, or local increase in the elevation of the water surface (Gippel (1995), Gippel *et al.* (1996a), Gippel *et al.* (1996b)). Therefore, LWD are associated with abrupt steps, indicating localized energy dissipation by LWD drag. Manga and Kirchner (2000) showed that, when the Froude number is small, the shear stress due to woody debris is

directly proportional to the local afflux. Moreover, they reported that half the drop in the water surface elevation through the surveyed reach occurred in steps associated with LWD. In other words, half the total dissipated energy – or half the total shear stress – was caused by LWD, a result that was also furnished by direct measurements of drag on woody debris.

In their experiments, Gippel *et al.* (1996b) measured that 15 % of the total afflux was caused by the largest item (out of a total of 95 items of debris) and the ten largest items accounted for 57 % of the total afflux. Now, relating these results with those of Manga and Kirchner (2000) might suggest that more than half the total resistance due to woody debris be caused by the ten largest items. This might further suggest that more effort should be directed toward an accurate modeling of local flow around the largest items and that the latter should be geometrically well represented and maybe included into the stream morphology. In this respect, Shields and Nunnally (1984) suggested that logjams large enough to have a damming effect be incorporated in backwater profile computations as geometric elements in the channel boundaries rather than being treated as roughness components.

This last recommendation is important for it consists of a deviation from the global approach associated with the flow resistance equations described earlier. The latter may be adequate for the hydraulic management of the stream – such as LWD removal or introduction and its global effect on resistance – but do not represent any local effect – which is believed to be the most significant on an ecological point of view.

Velocity distribution

A closer look at the flow patterns in the neighborhood of LWD would not only help obtain a more accurate description of the flow on the stream-scale, but it would also assist the prediction of aquatic development of fauna and flora. Indeed, hydraulic diversity created and maintained by debris enhances fish species diversity by providing habitat, through a range of flow conditions, for a variety of species and age groups. Dead-water zones provide areas for resting and for refuge during high flow conditions and low-velocity zones furnish a concentrated source of food (Sullivan (1987) in Gippel (1995)).

Moreover, the knowledge of precise values of depth and velocity at numerous points within the study reach is required for Instream Flow Needs (IFN) assessment techniques. One of the most widely used IFN assessment models in North America, the Physical Habitat Simulation System (PHABSIM), utilizes these hydraulic parameters as input variables to produce relationships between streamflow and usable habitat area for different life stages of varying fish species (Ghanem *et al.*, 1996). It is expensive to perform measurements in the field in order to obtain those parameters so that a hydrodynamic model that would provide these input variables is desirable.

As it was mentioned earlier in this paper, very few studies have focused on the local flow pattern around woody debris and most hydraulic research has been directed toward determining the global effect of LWD – traditionally on flow resistance – given the density of debris. However, using field measurements on a sand-bed stream reach in East Germany, Mutz (2000) assessed the local flow patterns and turbulence in the neighborhood of woody

debris. His study showed that the flow pattern was clearly controlled by the wood. Mutz turned his attention on two types of woody debris, depending upon its height relative to the stream bed. Woody elements elevated above the stream bed deflected the flow and locally caused strong secondary currents and high turbulence. Woody debris resting directly on the stream bed determined the roughness of the latter. More will be said in the next section about wood as roughness elements.

The localized effect of elevated wood can also be seen for the vertical velocity distribution. In a section intersecting big log, the flow was directed towards the stream bed and the vertical velocity gradient in the free flowing water could become reversed. These results suggest that:

1. the local flow around elevated woody debris be inherently three-dimensional. As a consequence, any depth-averaged two-dimensional modeling will fail to represent the local hydraulic diversity associated with elements of wood;

2. elevated woody debris be generator of high turbulence intensities. We may therefore legitimately expect this type of debris to be the cause of energy dissipation through turbulence in the first place.

LWD as roughness elements

The hydraulic effects of LWD have been reviewed on the global scale given a certain density of woody debris (i.e., effects on flow resistance) as well as on the local scale, generally around single elements of debris. However, LWD also influences the flow on an intermediate scale, depending upon the pattern of woody elements lying on the stream bed. In his review of woody debris hydraulics, Gippel (1995) refers to this situation as *multiple roughness elements*.

According to the nomenclature introduced by Morris (1955) and subsequently used by Davis and Barmuta (1989) and Young (1992), we may define three types of flow over roughness elements based on the roughness index, which is the ratio of horizontal roughness spacing λ to roughness height h. The three types of flow are depicted in Figure 1.3.

When the roughness elements are far apart, they act as isolated bodies on which are exerted drag forces by the flowing fluid. The wake zone and vortex-generating zone at each element are completely developed and dissipated before the next element is reached. The apparent friction factor would therefore result from the form drag on the roughness elements in addition to the bottom friction between elements. This type of flow is termed *isolated-roughness flow*.

The so-called *skimming flow* (or quasi-smooth flow) occurs when the elements are so close together that the flow skims over their crests. In the groove between the elements, there will be regions of dead water containing stable vortices. According to Morris (1955), much of the energy loss can be attributed to the maintenance of the groove vortices.

An intermediate situation develops when the distance between each element is approximately equal to the length of the wake generated by each element, in which case *wake-interference flow* occurs and considerable turbulent mixing is generated.

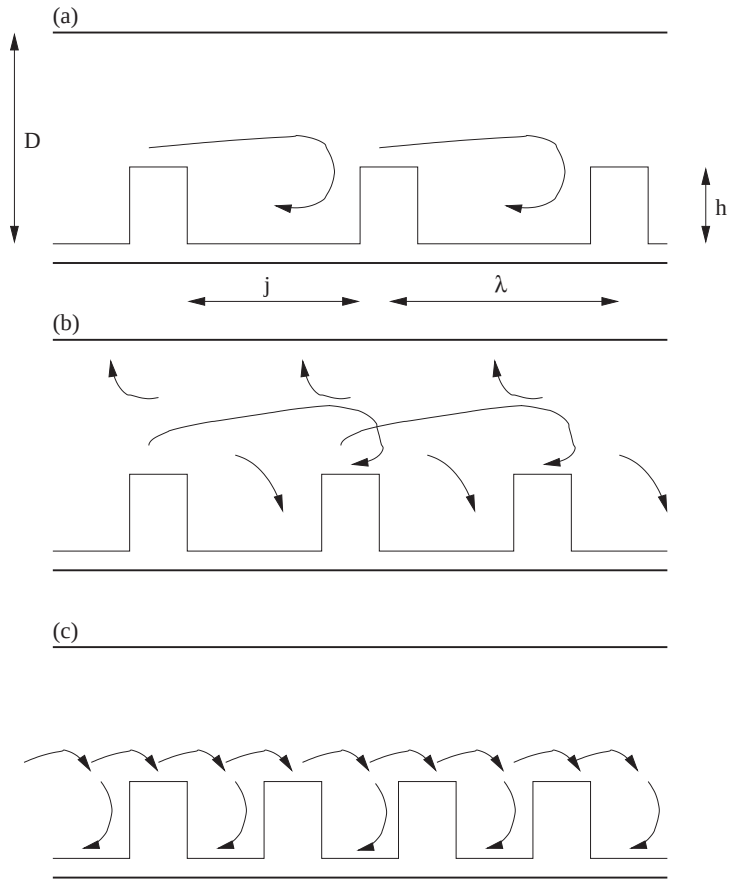

Figure 1.3: The three types of flow based upon roughness element geometry. Redrawn from Young (1992). (a) Isolated roughness flow. (b) Wake interference flow. (c) Skimming flow.

To define threshold criteria between those types of flow, three parameters are of importance: the roughness height (h), the roughness spacing (λ) and the groove width (j). When the roughness index λ/h is large, isolated-roughness flow will occur whereas when λ/h is very small, skimming flow will be present, provided that the roughness height be of reasonable value relative to the depth D. In this respect, Davis and Barmuta (1989) and Young (1992) noted that chaotic flow occurred for roughness height such that $D \leq 3h$. Under such conditions, flow structure is very complex and near-bed velocities are determined by the shape of the local flow boundary. In chaotic flow, the entire flow is affected by the geometry of the bed and energy losses are high.

The distinction between isolated-roughness flow and wake-interference flow can be made when the wake behind the roughness element just reaches the next roughness element. This transition is primarily affected by the roughness spacing λ. By equating the expressions for the frictional resistance in the two types of flow, Morris (1955) derived an equation (his Eq. (27)) for determining the critical value of transition λ_c:

$$\frac{\lambda_c/h}{C_d\left(1-\frac{ns}{P}\right)} = \frac{67.2/100}{\left(2\log y/\lambda_c + 1.75\right)^2} - 1, \tag{1.12}$$

where C_d is the roughness element drag coefficient, P is the cross-stream wetted perimeter, n is the number of elements across a section, s is the cross-stream groove width and y is the depth of water above the roughness element.

As noted by Morris (1955), a criterion to differentiate between wake-interference flow and skimming flow cannot be set up in a similar manner – that is by equating two expressions of frictional resistance – because the latter move away from each other rather than converge as λ approaches the critical value. Thus, there is likely to be a sudden change, occurring when the stable vortex in the groove gives way to the typical flow-separation phenomenon. Wake-interference flow is likely to appear when the groove width j is much larger than the depth D, in which case the vortex will adhere to the upstream face of the groove and the stream will flow over and down the vortex against the downstream groove face.

It should be pointed out that, from a biological perspective, it is the threshold between skimming and wake-interference flow that is of far greater importance, inasmuch as it indicates a change from stable, relatively sheltered conditions within a groove to an unstable, more turbulent flow regime Young (1992).

The above discussion shows that effects of roughness elements on the flow field may be reasonably well predicted if we assume that

1. the spatial distribution of woody debris on the bed may be retrieved;
2. the geometry of single elements of wood is fairly well known;
3. the spatial distribution is uniform, without which the previous results might become irrelevant. (We may relax the last statement by assuming that a *patchwise* uniformity may be sufficient for the applicability of the results).

In the realm of stream restoration, the work by Morris (1955) is useful and of direct applicability because it provides information as to how arrange woody debris in rivers to minimize resistance for a given desirable debris volume (for ecological purpose). In this very situation, people have control on the distribution as well as on the geometry of single elements. However, in a reversed situation in which the stream under study presents LWD randomly distributed by nature, this does not hold true. It then becomes indispensable to devise a technique to assess the distribution and geometry of LWD. As we will see later, it seems that a systematic approach to evaluate the distribution of LWD is yet to be found and most people have employed rather archaic methods to do so (e.g., close-up photographs, a method that could be invalidated in case of high turbidity).

To finish this section, we ought to mention a study by Nowell and Church (1979). They extended Morris (1955)'s approach by classifying flow types according to the planform density of roughness elements (that is, the ratio of total plan area of roughness elements to total plan area of channel). Skimming flow occurred at densities of 0.125-0.083, wake-interference flow occurred at densities of 0.063-0.045 and isolated-roughness flow required a density as low as 0.02.

LWD and dimensionless numbers

A somewhat different perspective on LWD is described in Kemp *et al.* (2000). They established a link between so-called *functional habitats* (biologically defined habitat units) and *flow biotopes* (hydraulically defined habitat units) using Froude number. Functional habitats are objectively defined habitat units, made up of substrate or vegetation types, which have been identified as distinct by their invertebrate assemblages. Fifteen of the 16 functional habitats were found to be distributed with Froude number in a non-random fashion, woody debris being the exception. This information may prove useful for stream rehabilitation projects insofar as hydraulic dimensionless numbers, such as Froude number, can be manipulated through changes to channel morphology in order to obtain desired habitat heterogeneity (Kemp *et al.*, 2000).

The lack of correlation between woody debris and Froude number may misleadingly suggest that flow characteristics not influence the pattern (distribution and whether woody debris is present or not) of LWD. However, this conclusion might conceal other potential causes to this lack of correlation, as mentioned in Hodges (2002):

1. Other hydraulic variables, such as the Reynolds number or turbulent intensities, may be significantly correlated to functional habitats made up of LWD.

2. Froude number is important but its measurement in the presence of LWD was faulty (because strongly affected by secondary currents and/or fluctuating velocities associated with turbulence).

3. There are some scales of LWD for which no correlation exists between flow type and habitat. (For some scales – in particular very large pieces of wood –, it might be more successful to consider woody debris as being part of stream morphology rather than functional habitat).

1.1.3 Effects of LWD on stream morphology

Although this review is intended to mainly center on the hydraulics of LWD, for the sake of completeness and because modifications of stream morphology eventually affect the flow pattern – whether there is woody debris or not –, we should briefly review the effects of LWD on stream morphology. Following the suggestion of Harmon *et al.* (1986), the geomorphic roles of LWD can be grouped into effects on landforms and on transport and storage of sediment. A priori, modifications of landforms are more significant to affecting, in turn, the flow field whereas sediment transport is more likely to matter on an ecological point of view even if changes in local bed roughness – and thus flow resistance – are expected as well.

Many studies showed that pools were associated with the presence of large woody debris lying on the bed or partially spanning the channel with one end supported on the bank and the other on the streambed (Keller and Swanson (1979), Cherry and Beschta (1989), Mutz (2000)). A generic situation is delineated in Figure 1.4. LWD can increase pool frequency and variability in pool depths (Harmon *et al.*, 1986). In addition to locally scouring the stream bottom, erosion may also increase channel width as water is diverted around the obstruction (Keller and Swanson, 1979).

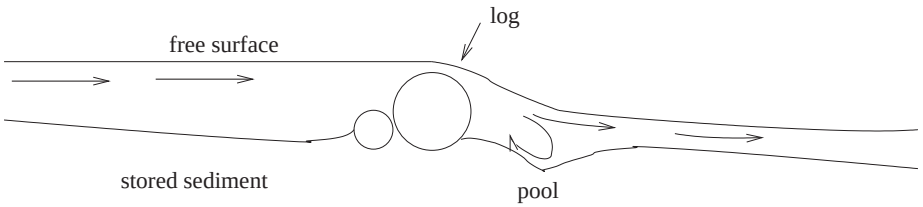

Figure 1.4: Idealized diagram showing concept of pool formation. Redrawn from Keller and Swanson (1979).

Even if stream morphology is affected, time scales of acting processes are much larger than that associated with streamflow features. As an example, the presence of LWD may deflect the flow toward the bank, thereby accelerating stream erosion. But, whereas changes in the flow field occur on short time scales, bank erosion happens on much larger time scales. Moreover, the acceleration of the latter is exclusively caused by diverted flow, hence advocating the need to study streamflow patterns in the first place.

1.1.4 Indirect effects of LWD

As suggested by the cause-effect relationships diagram in Figure 1.2, LWD indirectly affects stream ecology through changes in flow patterns (link V in the diagram) and stream morphology (link VI). Indeed, as we have seen, the presence of LWD creates regions of low-speed flow, which are preferred habitats or refuges for many fish species. Also, the diversity in pool distribution and depth has been proved to enhance fauna variety. Furthermore, sediment that is retained by LWD may contain organic matters that are beneficial to stream ecosystem. Now, as it was already mentioned earlier, there exists a close relationship between flow patterns and stream morphology (represented by link IV in the diagram). Although we do not intend to review these indirect relationships between LWD and ecology (namely

links IV to VI) – they were the topic of many studies in the past –, the aim of the above considerations was to support the proposition that stream hydrology is found at the center of those interactions and that a closer and detailed look at the direct relationship between LWD and flow patterns, without being the panacea, is likely to furnish many answers.

1.2 Objectives

Evaluation of flow resistance on a global scale (a zero-dimensional method) does not generally require any assessment of LWD distribution more accurate than that given by its planform density. The knowledge of flow resistance is of high significance when it comes to managing flow capacity. In particular, for regulated rivers, in which flow capacity is to be maximized, the optimum debris loading will be the minimum required to maintain ecological integrity. On the other hand, flow resistance is very likely to be poorly correlated to local stream ecosystems. The so-called field of *ecohydrology*, linking channel hydraulics and morphology (Kemp *et al.*, 2000), is chiefly concerned with local in-stream physical effects. Ecohydrology therefore becomes relevant when local in-stream measurements – of flow types and morphological features – are available, or provided by a hydrodynamic model. Not surprisingly, in this respect, the two-dimensional finite element model of physical fish habitats developed by Ghanem *et al.* (1996) appeared to be significantly better than a one-dimensional approach, such as the application of HEC-2. Their results strongly encourage the utilization of such model dimensionality – with adequate subgrid scale parameterization to account for the presence of LWD – to predict physical habitat distribution in streams with woody debris.

Nonetheless, 2D models require detailed bathymetry surveys as well as methods allowing for the identification of LWD in the data set. The latter requirement is crucial in the modeling process because it allows for discerning what would be true bathymetry behind that *polluted* by LWD. Furthermore, knowing the locations of LWD is useful for aquatic habitat analysis.

The objective of this research is to develop a systematic approach to identify LWD within a bathymetry survey data set in order to produce two outputs:

1. Bathymetric data set devoid of LWD, ready to use in 2D modeling (e.g., for interpolation).
2. A set of LWD locations, ready to use in aquatic habitat analysis.

This thesis describes the steps taken to achieve this objective.

Chapter 2

Methods

Over the past decade, two-dimensional (2D) hydraulic models of rivers and streams have been supplanting one-dimensional (1D) models for use in aquatic habitat analysis (CITE). The increase of model dimensionality allows better representation of the spatial structure of the flow depth and velocity that affects habitat availability, while simultaneously reducing the need for extensive field data over multiple river discharge rates for model calibration (CITE). However, there does appear to be a conservation of difficulty. While requiring less flow data from the field, the 2D models require more detailed bathymetric surveys. Furthermore, the survey result must be interpolated to the 2D model grid (see Figure 2.1), so the complex relationship between the survey resolution, model resolution and method of interpolation affects the final accuracy of the model bathymetry (CITE paper on MEBAA that Barney and Tim are working on?). For smooth bathymetries, there is little difficulty in developing accurate translations from survey data to model; however, in rivers with significant bottom structure (e.g., LWD, Figure 1.1), simple data averaging and interpolation methods may lead to misrepresentation of the bottom bathymetry (see Figure 2.2) that can distort the depth and velocity results of a model. In this thesis, we examine systematic methods for identifying LWD in single-beam echo sounder data so that the river bathymetry (rather than the LWD) can be appropriately interpolated to the model grid.

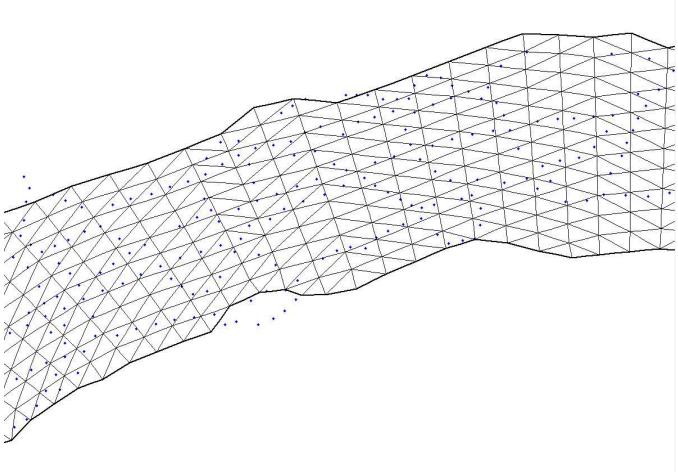

Figure 2.1: Surveyed bathymetry data points (dots) must be interpolated to the finite element mesh.

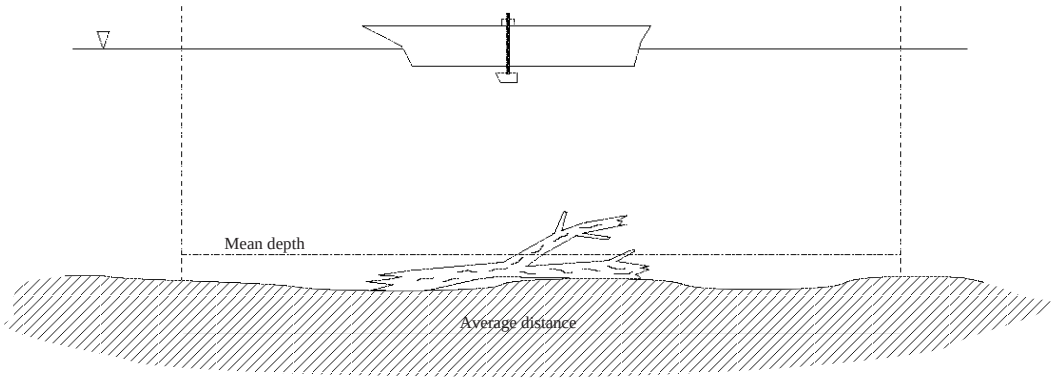

Figure 2.2: Distortion of bathymetry data due to the presence of large woody debris.

As a part of an aquatic habitat analysis for the Sulphur River, Texas (Osting *at al*, 2003), the Texas Water Development Board (TWDB) conducted a fine-scale bathymetric survey of a 1.36 km river reach on the mainstem Sulphur River. Streamflow in the Sulphur River is generally from west to east and drains approximately 9300 square kilometers. Data of hydraulic site located just north of IH-30 and just west of the US-259 bridge that crosses the river is under examination in this paper. The river bathymetry was surveyed using an echosounder (Knudsen Engineering's 320BP High Frequency 200 kHz Portable Echosounder) recording an average of nine depth measurements per second, while the boat position was recorded only once per second using a differential Global Positioning System (GPS). The TWDB used a single-frequency (L1) Trimble ProXRS GPS receiver with real-time satellite differential correction (DGPS) service provided by Omnistar. The boat speed (based on GPS data) averaged 1.4 m s^{-1} with a standard deviation of 0.5 m s^{-1}. Previously, TWDB bathymetric surveys used the average of the nine depth measurements taken around each GPS data point, giving an effective survey resolution along the boat track of 1.5 m. However, as LWD may have width scales on the order of 10 cm, it follows that averaging the sounding data over a GPS position will distort the computed bottom boundary as illustrated in Figure 2.2. Using a linear estimate of the boat velocity from GPS data and distributing the depth measurements uniformly along this track, the survey resolution is approximately 16 cm. As shown in Figure 2.3, this higher-resolution bathymetry shows spikes that are significantly moderated in the averaged bathymetry, and appear to distort the smoothness of what might be expected to be true bathymetry.

It was impractical to provide direct physical confirmation of the correlation between the data spikes (e.g., Figure 2.5) and LWD at the high flow rates under which the Sulphur River bathymetric surveys were conducted. While it is reasonable to infer such correlation based on the photographic evidence of emergent LWD at low flow rates (e.g., Figure 1.1), to improve our confidence in this inference we conducted a separate field test to examine the performance of the depth sounder over a known piece of LWD. On April 2, 2003, we located a piece of emergent LWD in the Guadalupe River of Central Texas (see Figure 2.6) that had a submerged section approximately 60 centimeters below the water surface. To provide controlled and repeatable data collection over the LWD and across the river during the relatively high flow rate period, a rope was stretched across the river and the boat was hand towed at speeds of 0.4 m s^{-1} and 0.6 m s^{-1}, which is somewhat slower than the 1.4

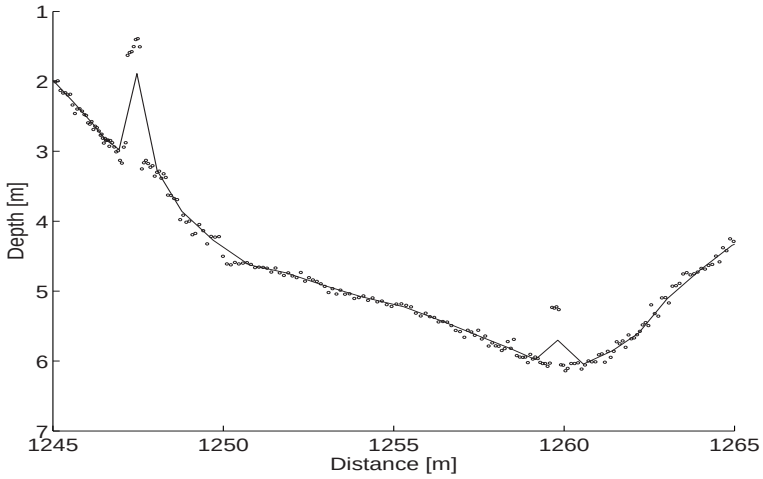

Figure 2.3: Distributed depth measurements along boat track are represented by points while the solid line is the averaged bathymetry.

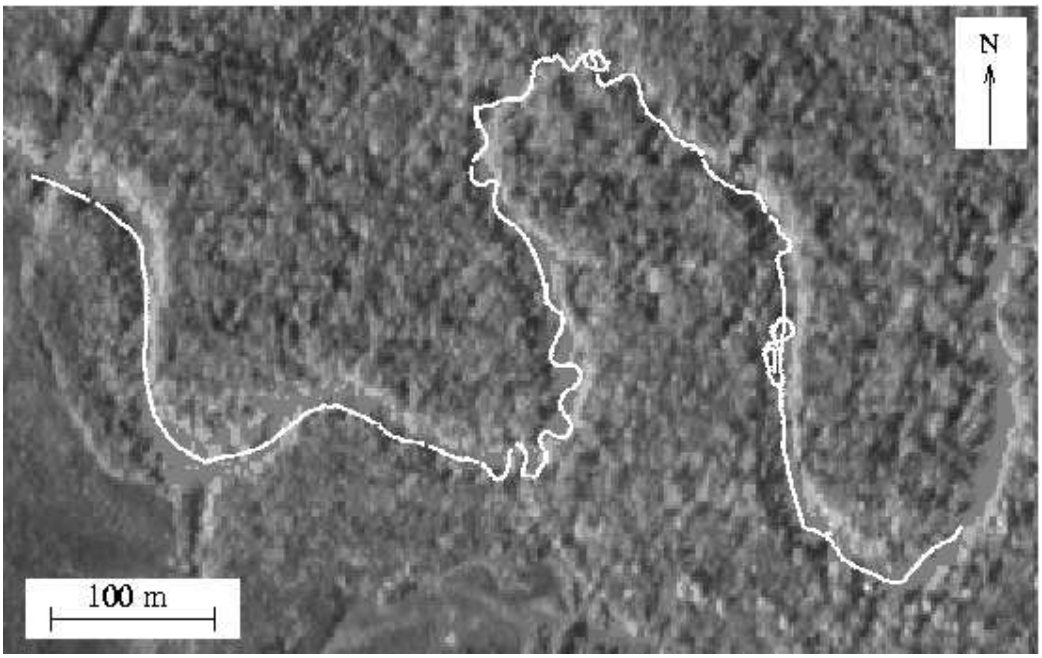

Figure 2.4: One of the boat tracks for the Sulphur River bathymetric survey conducted on May 2001 by TWDB between (33° 18' 31.23" N, 94° 43' 37.57" W) and (33° 18' 24.18" N, 94° 43' 09.70" W).

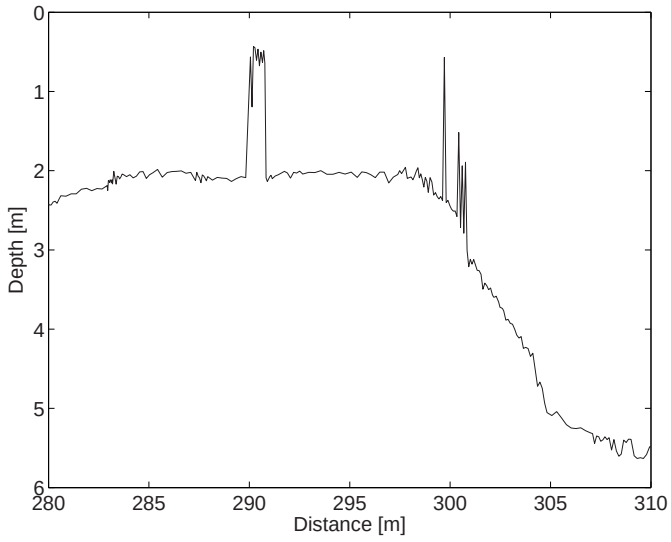

Figure 2.5: Section of Sulphur River bathymetry containing spikes. The average distance between depth measurements is 16 cm and for the purpose of clarity, linear interpolation has been performed.

m s^{-1} speed used in the Sulphur River survey. The results of higher speed surveys can be estimated by sub-sampling the data sets. It is clear from Figure 2.7 that the signature of the LWD in the Guadalupe River data set is similar to the spikes seen in the Sulphur River (Figure 2.5).

Figure 2.6: Submerged piece of woody debris in Guadalupe River.

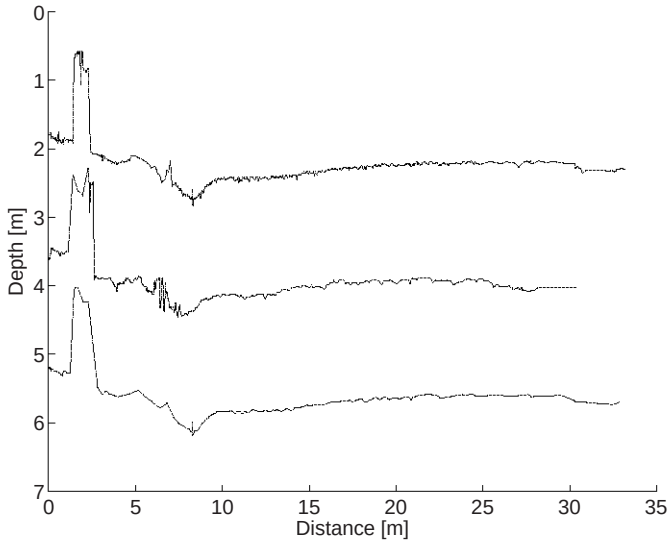

Figure 2.7: Surveyed cross-section of Guadalupe River over submerged piece of LWD (represented by the spike on the left side). Top and middle graphs show profiles obtained at different boat speeds (0.4 m s^{-1} and 0.6 m s^{-1}, respectively). Bottom graph is a decimated version of the top graph, sub-sampled at every 4^{th} data point.

Chapter 3

Statistical techniques

The aim of this chapter is to evaluate what statistical analysis techniques can yield in terms of LWD spatial distribution.

3.1 Moving average

Computing the mean depth for each distinct GPS position, we obtain profiles that are smoothened out as shown in Figure 3.1. The title of this section might be misleading inasmuch as, strictly speaking, the process of computing the average per GPS position does not equate a moving average in our case. Indeed, the number of depth soundings included in the mean depth calculations varies from six to ten, with 86% of mean depths calculated from sets containing nine depth soundings. For this process to be a moving average, each successive mean depth ought to be computed from sets containing exactly the same number of depth soundings. We will, however, continue to refer to it as a moving average.

Examination of these profiles allows for making the following comments. Computing the mean depth for each distinct GPS position inevitably leads to the disappearance of some spikes when these only consist of one or two depth soundings. On the other hand, broader spikes linger after averaging. This dual situation can be seen in Figure 3.1, especially graph (d), where only spikes around distances of 1250 m and 1260 m remain after averaging while spikes situated at positions 1180 m and 1225 m completely vanish. Similarly, examining graph (b) permits to witness the absence of spikes in the mean depth profile at distances of 300 m and 370 m. Yet, severe spikes can be seen in on the dotted line at the same locations.

One would then legitimately enquire as to why such an averaging process – accompanied with a loss of valuable information – would take place. The answer lies within the realm of numerical computations. In hydrodynamic modeling of rivers, grid resolutions very seldom attain values below one meter so that, eventually, bathymetry data will be interpolated at computational nodes that are of the order of one meter apart (for the finest grid resolutions). Consequently, there is no possibility for keeping intermediate depth soundings per se (16 centimeters apart, on average), yet crucial need for these coalesced data (the mean depth in our case) to contain as much information associated with intermediate depth soundings as possible. This consideration should be the principle criterion upon which any selection of statistical or spectral technique is made.

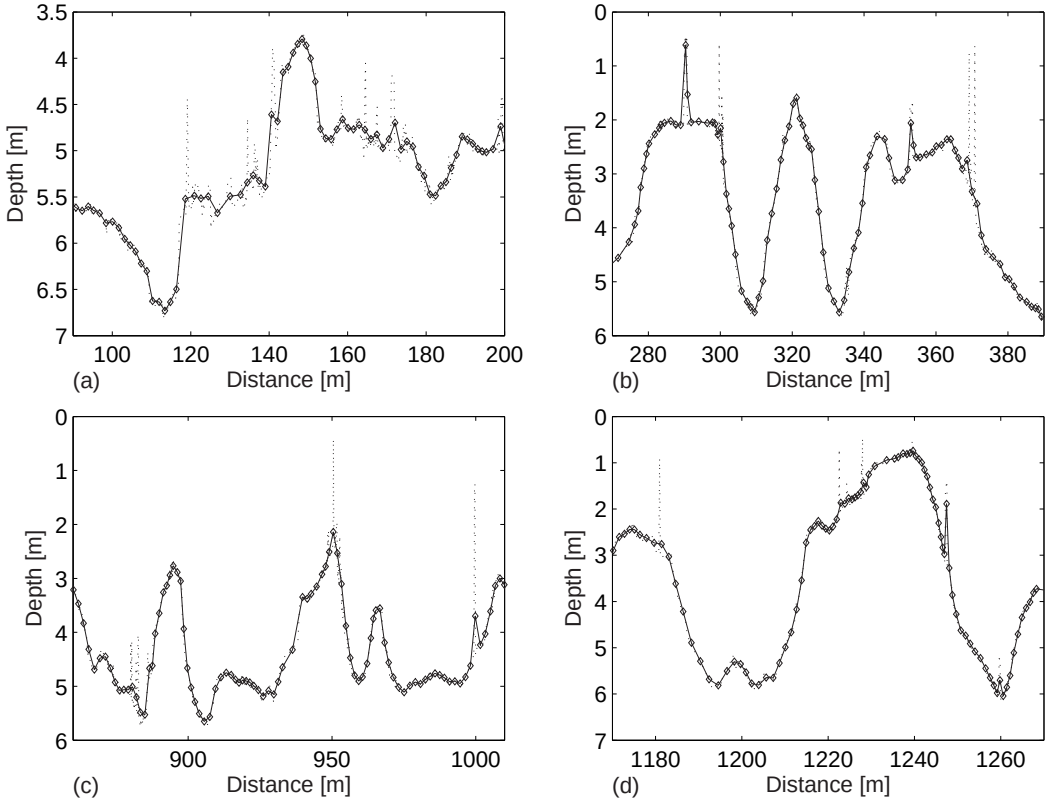

Figure 3.1: Selected sections of bathymetry data: diamonds represent mean depths for each distinct GPS position while the dotted line is the bathymetry including all depth measurements.

The moving average, as it was presented, conducts to a loss of extremely valuable information. Most spikes simply get absorbed into the calculation of the mean depth, leading to a much smoother profile. It should be pointed out, however, that some spikes remain visible in the smooth profile. This occurs when the spikes are formed by a large set of depth soundings. In that situation, the mean depth is closer to the would-be woody debris depth than the streambed depth. On the other hand, when the spike is made of one or two depth soundings, the mean depth reaches a value close to that of the streambed depth. This dual situation is illustrated in Figure 3.2.

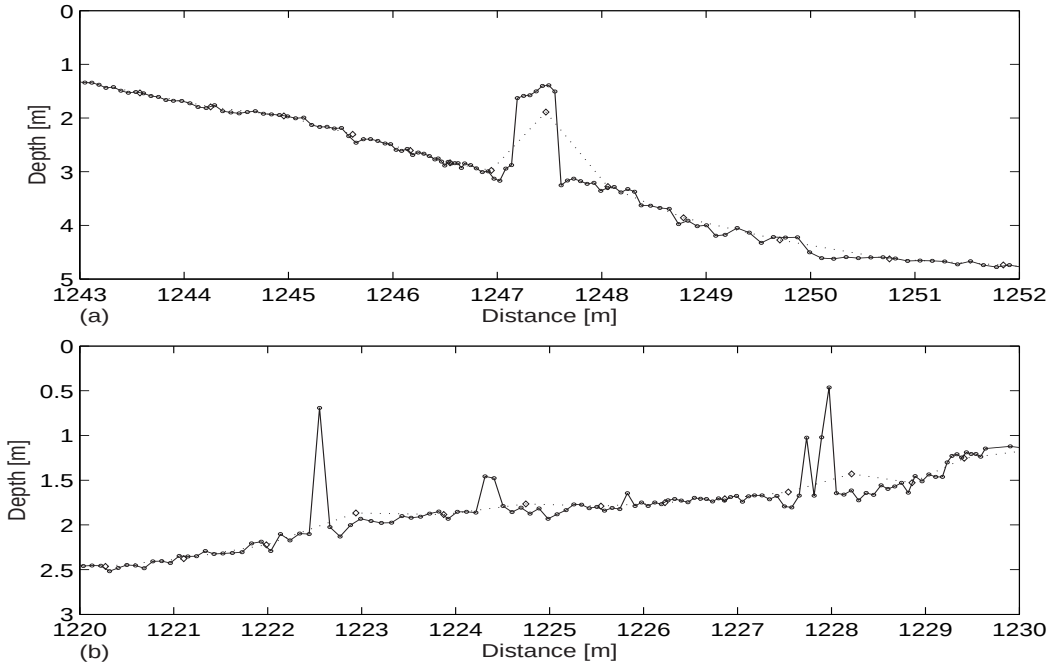

Figure 3.2: Influence of the number of depth soundings (represented by ∘) forming the spike on the calculation of the mean depth (represented by ◊). (a) A spike caught by seven depth soundings leads to a high mean depth, close to the spike depth itself. (b) Spikes made of one or two depth soundings do not result in an observable disruption of the smooth profile.

For the sake of completeness, the same comparison analysis may be performed on the Guadalupe River survey results. Namely, a superimposition of the original profile (containing all depth soundings) and the mean depth profile. As it was previously suggested, slow-pace river crossings should lead to spatially wide LWD signatures, that is including a significant number of depth soundings. A corollary is the subsistance of a spiky signature in the mean depth profile. This is illustrated in Figure 3.3.

3.2 Moving average and standard deviation

The aforementioned scenario suggests that some improvement be made to this simple averaging technique. Together with the mean depth, it turns out to be informative to compute the standard deviation of the set of depth soundings. A large standard deviation is expected to result from sets containing spikes shaped by one or two depth soundings. This

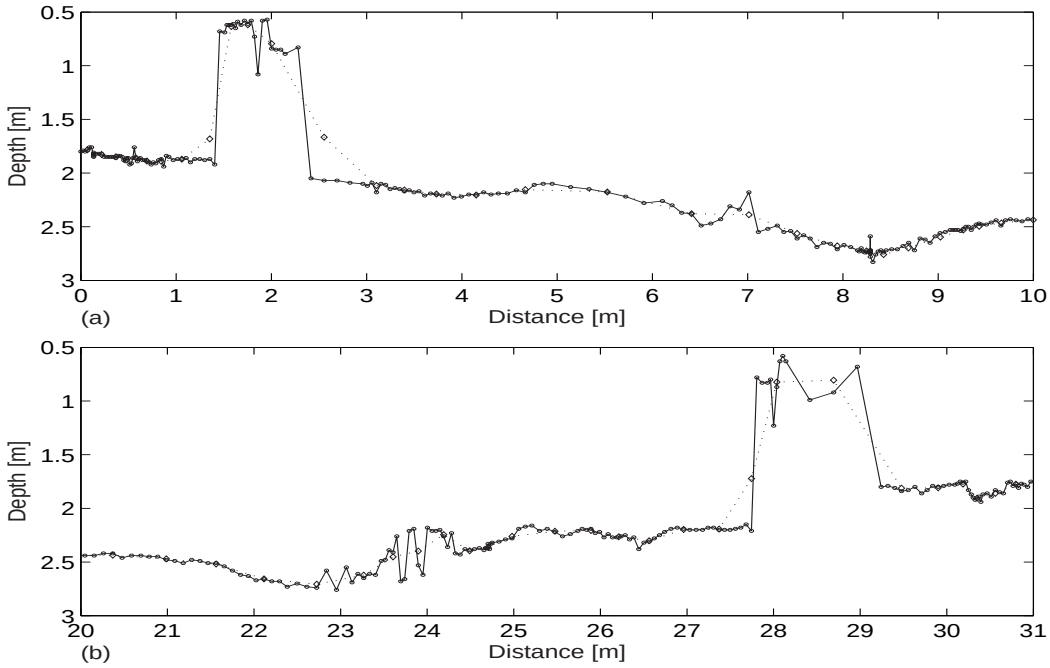

Figure 3.3: For the pilot study in the Guadalupe River, slow-pace crossings lead to spatially-wide LWD signatures (depth measurements are represented by ∘). The mean depth profile (dotted line and ◊) reflects this peculiarity. (a): Depth measurements taken at a slow pace from right bank to left bank. (b): Depth measurements taken at a faster pace from right bank to left bank.

expectation proves true and previously-studied bathymetry profiles are shown in Figure 3.4. A close look at graphs (b) at distances 300 m and 370 m and (d) at distances 1180 m and 1225 m permits to establish the usefulness of computing the standard deviation. Indeed, these locations do not feature any spike in the mean depth profile but a large standard deviation – indicative of the presence of spikes in the original profile – is clearly visible.

The combination of mean depth and standard deviation allows for keeping some information regarding the location of spikes. However, width and depth of LWD still remain unknown. The averaging process, which gives rise to a bathymetry that neither is totally smooth nor contains all its original spikes, is rendered useless by the latter property. One would rather wish to work with a bathymetry that is completely disposed of disruptive spikes, yet still know where those spikes lie within the profile in order to model LWD effects properly.

3.3 Semivariogram

The semivariogram approach as been widely applied in mining geostatistics for which it was originally developed (Journel and Huijbregts, 1978). The semivariogram has been used as a basic statistical method for investigating roughness properties of bed profiles obtained from field work and laboratory experiments (Oliver and Webster (1986), Robert (1988) and Robert and Richards (1988)).

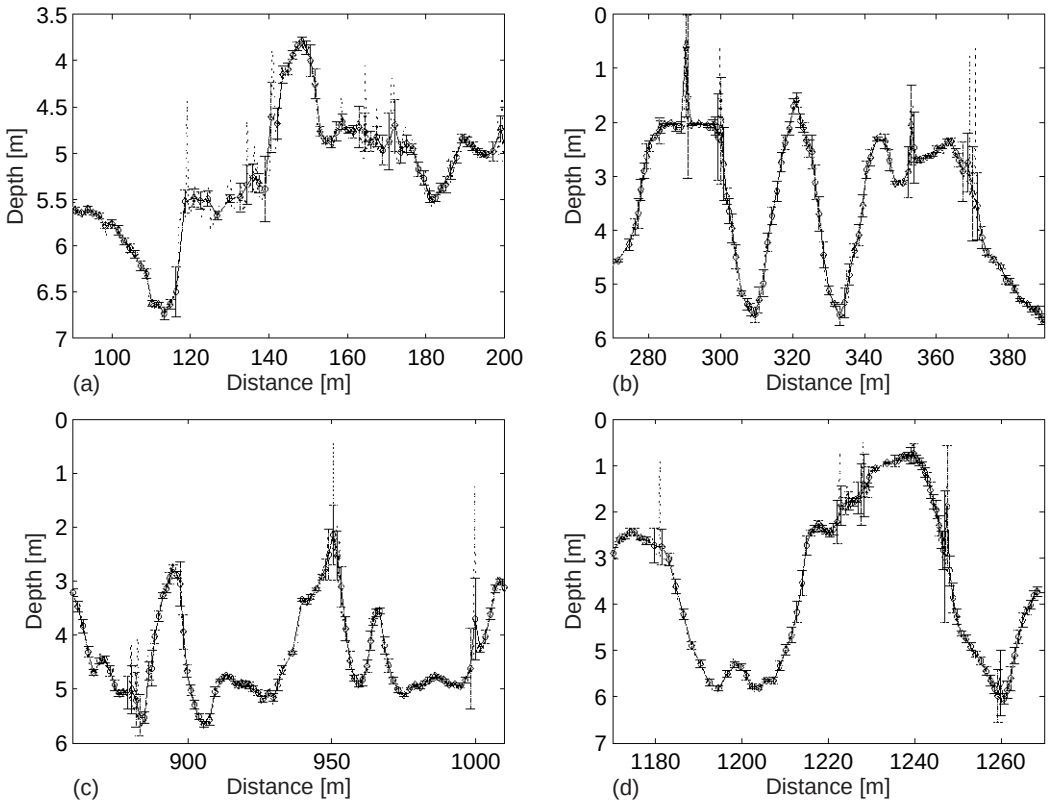

Figure 3.4: On each graph, the dotted line is the bathymetry profile represented by all depth soundings. The diamonds are the mean depths, as in Figure 3.1. Error bars indicate standard deviations.

Under the stationarity hypothesis[1], the semivariogram function $\gamma(h)$ takes the following form Journel and Huijbregts (1978):

$$2\gamma\left(h\right) \doteq \mathrm{Var}\left[\mathcal{Z}(x) - \mathcal{Z}(x+h)\right] = \mathrm{E}\left[\left(\mathcal{Z}(x) - \mathcal{Z}(x+h)\right)^2\right] \tag{3.1}$$

where h is the lag and $\mathcal{Z}(x)$ is the random variable of interest. In our situation, the random variable is the stream depth and the stationarity hypothesis translates to assuming that the semivariogram function does not depend upon the locations defining the lag h but rather only depends upon h itself. Such an assumption renders statistical inference possible (Journel and Huijbregts, 1978).

In practice, only discrete samples are available and the empirical semivariogram $\hat{\gamma}(h)$, reads (Robert and Richards (1988) and Robert (1988))

$$2\hat{\gamma}(h) = \frac{1}{(N-h)} \sum_{i=1}^{N-h} \left[\mathcal{Z}(x_i + h) - \mathcal{Z}(x_i)\right]^2,$$

where N is the number of measurements.

The function $\gamma(h)$ summarizes the spatial variation in the data. It is customary to fit standard mathematical functions to sample semivariograms from which parameters can give insight into the nature of spatial variation of bed elevations. It should be emphasized, however, that the usefulness of this approach resides in its ability to extract information on larger-scale bedforms by bypassing erratic smaller-scale components. In this respect, we should not expect the semivariogram to yield valuable information on the spikes since its main purpose is to get rid of them. Keeping in mind that our utlimate goal is to find a way to separate – or filter out – all spurious spikes from an otherwise smooth bed profile, the semivariogram technique ought to be regarded as a way to evaluate to what extent such a filter affects larger-scale bedforms.

Due to boat speed variations, the bathymetry sampling interval ($\Delta\xi$) is not constant. In order to compute the semivariogram, the bathymetry was subsequently interpolated at a constant value of $\Delta\xi = 15$ cm for a total number of samples $N = 12{,}000$. The semivariogram is shown in Figures 3.5 and 3.6, using linear and log-log scales, respectively.

On both Figures 3.5 and 3.6, the thick dotted line is the empirical semivariogram calculated from the bathymetry samples and the dashed line is an exponential curve fit obeying the following law:

$$c\left[1 - \exp\left(-h/r\right)\right] \tag{3.2}$$

where $c = 0.92$ and $r = 7.73$ have been obtained by minimizing the residual sum of squares.

[1] A random variable is said to be stationary when its spatial law is invariant under translation.

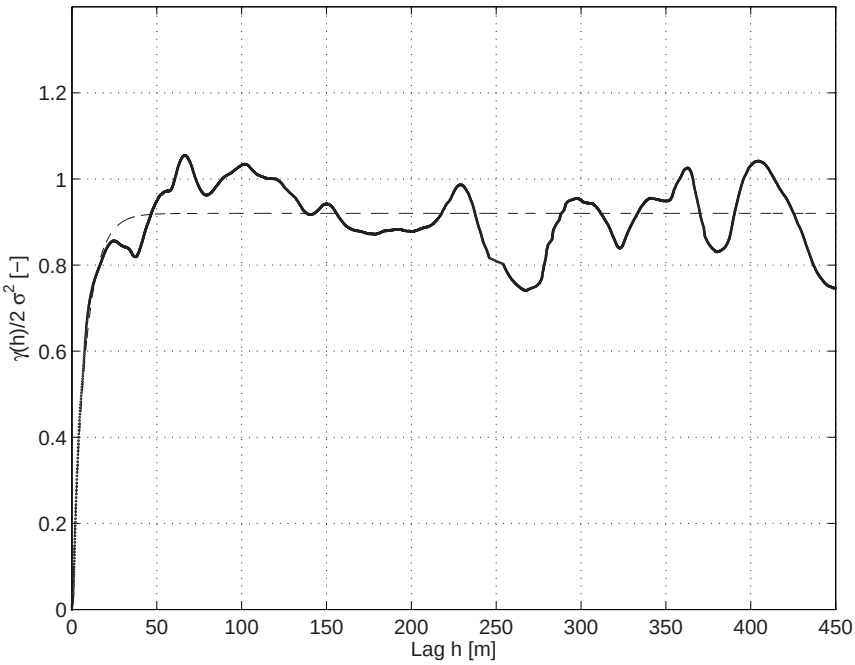

Figure 3.5: The thick dotted line represents the observed semivariogram. The exponential curve fit is depicted by the dashed line. The so-called *sill*, or a priori semivariance of the set, is found to be $c = 0.92$.

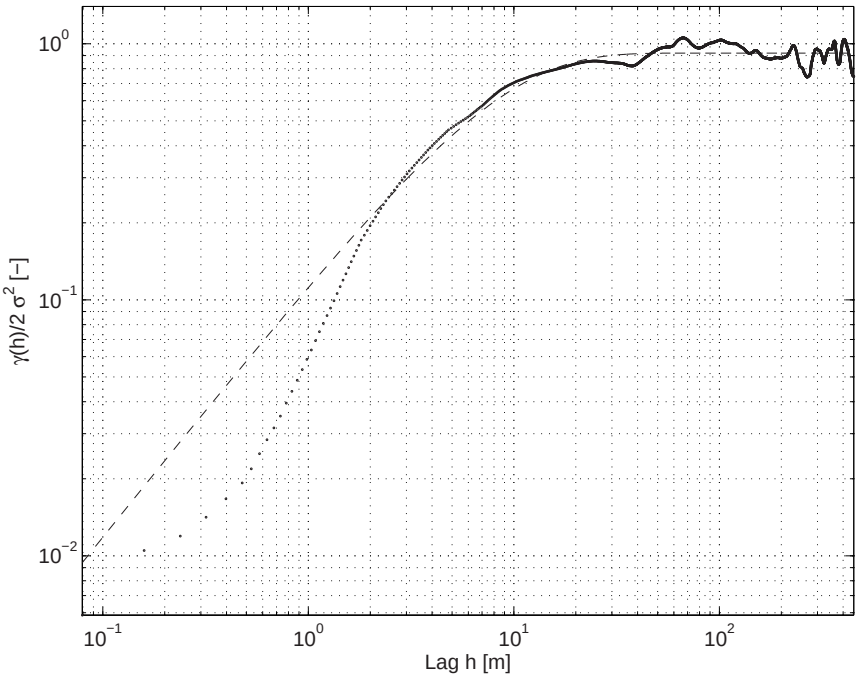

Figure 3.6: Same as Figure 3.5
on a log-log scale.

The semivariogram was calculated for lags up to a quarter of the series length (as recommended in Robert and Richards (1988)), i.e., up to 450 m. Beyond that point, results obtained from statistical inference may become questionable.

In his review of semivariogram general forms, Robert (1988) distinguishes between two general cases. Sand-bed profiles yield an unmistakable periodic oscillation at a more or less constant value of $\gamma(h)$ (identified as c in Figure 3.5). Semivariograms of gravel-bed profiles present totally different statistical properties. When plotted on a log-log scale, they present two distinct linear increases of $\gamma(h)$ with distance. Clearly, none of the semivariograms in Figures 3.5 and 3.6 present these properties. It is not clear why this semivariogram does not fall into one of the aforementioned categories but it might be caused by the coarseness of the sample. Indeed, Robert (1988) used a sampling interval of 5 mm, a factor of 20 smaller than ours.

3.4 Scale-space analysis

3.4.1 Presentation of the method

The scale-space filtering technique was introduced by Witkin (1983) for the analysis of signals and subsequently adapted by Bergeron (1996) to analyze stream-bed roughness. The modified technique provides a multiscale version of bed profiles that allows the identification of single roughness elements at all scales of observation. Two steps can be outlined:

1. **Successive applications of Gaussian filter on the original data**. The original signal is progressively smoothened by successively applying a Gaussian filter of mean $\mu = 0$ and standard deviation σ. The so-called scale-space image is a two-dimensional representation of smoothened versions of the original signal. Such a scale-space image is shown in Figure 3.7 after nine applications of the Gaussian filter of standard deviation $\sigma = 10$ m.

2. **Construction of fingerprint**. Each smoothened signal features peaks and troughs, some of them disappearing with increasing smoothening, as can be seen in Figure 3.7. A fingerprint is obtained by plotting the location of these features (peak or trough) against the smoothing level. The fingerprint associated with our previous example is shown in Figure 3.8 for 50 levels of smoothing, i.e., 50 applications of the same Gaussian filter.

The effect of the progressive smoothing can be visualized in Figure 3.7. Between approximately 220 and 400 meters, three main peaks are clearly distinguishable. On the sixth smoothened profile, between the same locations, those three peaks have collapsed into one main peak. A similar observation may be made for the bedform located between 1450 and 1550 meters. The first level of smoothing removes all details while preserving the general shape of the bedform. However, not only does Gaussian filtering simplify the signal by removing small-scale features, it also distorts it by broadening and flattening the remaining features (Bergeron, 1996). It should be pointed out that the distortion of the signal, inherent to Gaussian filtering, renders it useless for the very purpose of modeling. Yet, we should

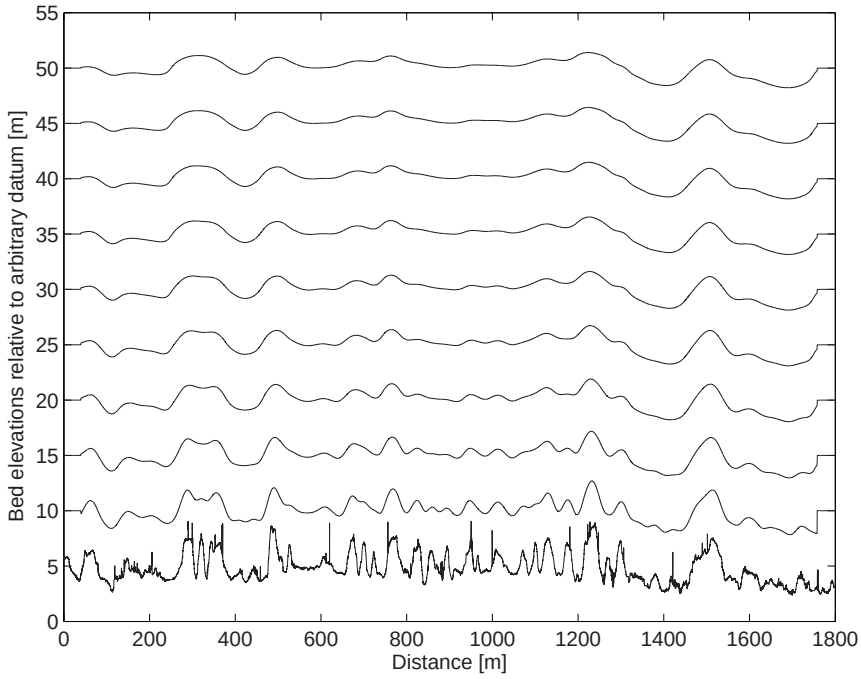

Figure 3.7: Scale-space image issuing from nine successive applications of a Gaussian filter of standard deviation of $\sigma = 10$ m. Original profile is at bottom.

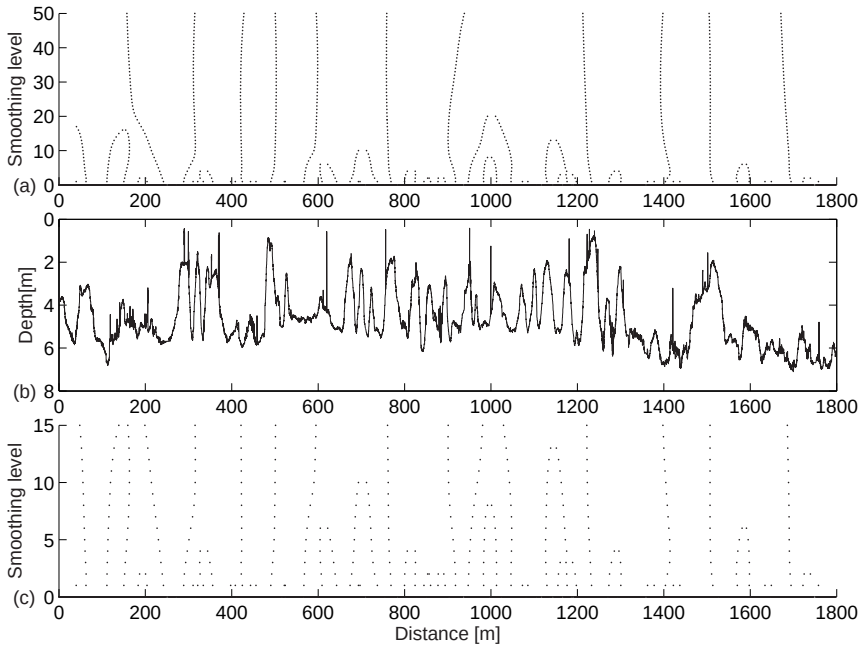

Figure 3.8: Fingerprint obtained by plotting the location of features (peak or trough) against the level of smoothing. Gaussian filter ($\sigma = 10$ m) has been applied 50 times. (a) 50-level fingerprint. (b) Bathymetry profile. (c) Zoomed fingerprint: 15 first levels.

be able to gather information on the structure of the signal at different scales, which is the objective of this technique.

The fingerprint contains the information required for a multiscale description of the signal. Each closed arch (visible for lower levels of smoothing in Figure 3.8) corresponds to a feature. Small arches are associated with small-scale features that disappear rapidly. Bigger arches correspond to larger scale features persisting over a wide range of scales (Bergeron, 1996). We may now examine how the fingerprint presented in Figure 3.8 relates to the scale-space image. Since only the first nine smoothened signals are shown in the scale-space image, we will investigate the fingerprint below a smoothing level of 10. The bottom graph in Figure 3.8 purposedly focuses on those lower smoothing levels. The small closed arch located at 1000 meters is to be related with the small bedform that can be seen in the original profile in the scale-space image. In the latter, it may be observed that this peaky feature persists for seven levels of smoothing, after which, it is totally flattened and collapses into a larger scale feature. Now, in the fingerprint, the arch closes at a smoothing level of eight. This exhibits coherence with the scale-space image. Two additional similar examples can be spotted around the locations 700 meters and 1600 meters. In the former, the three peaks first coalesce to form a larger, flattened, bedform, which eventually collapses into a new larger scale feature. Finally, as we may expect by looking at the original signal at the bottom of the scale-space image, a major bedform is located at 1500 meters. The ninth smoothened signal still clearly features this bedform. Moreover, as illustrated in the top graph of Figure 3.8, its persistence – at least up to the 50 levels of smoothing – is characterized by the vertical dotted line at 1500 meters. The same conlusion can be made regarding the persistence of the three big peaks around 300 meters.

The previous experiment – with a filter of standard deviation $\sigma = 10$ m – proves useful to collect information on relatively large-scale features. Nonetheless, no valuable information on the structure of sharp spikes – our interest – is available. It is closely related to the utilization of an inadequate standard deviation. All spikes disappeared after the first level of smoothing, thereby rendering arch formations impossible. This outcome suggests the use of a smaller standard deviation.

3.4.2 Implementation

Strictly speaking, the Gaussian filter is given by

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\left(x-\mu\right)^2 / 2\sigma^2\right). \tag{3.3}$$

Filtering consists in convolving the original signal with the Gaussian filter :

$$\begin{aligned} F(x) &= f(x) \star g(x) \\ &= \int f(u-x) \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\left(u-\mu\right)^2 / 2\sigma^2\right) du, \end{aligned} \tag{3.4}$$

where the integral is computed over the range of $f(x)$.

The bathymetry data that we are using is discrete and the above theoretical framework must be adapted accordingly. Several issues must be taken care of in the implementation. Since the filter is symmetrical, the discrete convolution is equivalent to replacing every sample by a weighted average of the bed profile over the width of the Gaussian filter, centered at the sample under consideration. In theory, the width of the filter is infinite so that a practical width must be selected. Following the recommendation of Bergeron (1996), a filter halfwidth 4σ was used. As a result, the filter could not be applied to bed elevations situated at the extremities of the dataset. The sampling interval of our dataset is not constant so that, for each bed elevation, the number of samples comprising the weighted average varies. This leads to some computational inefficiency – the filter width must be determined for each bed elevation – but interpolating the original profile to produce constant sampling intervals would inevitably lead to distortion of sharp spikes. The final step consists in the identification of peaks and troughs within each filtered profile. The whole procedure is performed by a C program, whose output is a series of vectors containing the locations of features for each level of smoothing. Plotting such information is straightforward in Matlab.

3.4.3 Applications

As previously suggested, a new experiment with a smaller standard deviation must be effectuated. The mean sampling interval being 16 cm, a standard deviation of $\sigma = 20$ cm has been chosen. This will ensure that all bed elevations within 80 cm of a given sample be included in the calculation of the associated weighted average. The fingerprints associated with bathymetry sections presented earlier in this paper – namely bathymetry sections of Figure 3.1a-d – are presented in Figures 3.9 through 3.12 and are discussed below. Gaussian filtering has been applied 200 times. Each figure features three graphs. The center one represents the bathymetry profile being analyzed while the top and bottom plots are fingerprints. The top one shows all 200 smoothing levels and the bottom graph zooms on the 50 first smoothing levels, which allows for identifying smaller scale arches.

Whereas fingerprints seem to be an adequate technique to identifying large-scale bathymetry features, a lack of coherence appears when it comes to spotting extraneous spikes. In Figure 3.9), we may put three spikes under scrutiny. They are located right before 120 meters, at 140 meters and right after 170 meters, respectively, and are indicated by arrows. The first spike is shaped by one sounding but no arch is associated with it. Instead, at this location, the fingerprint is characterized by a quasi-vertical line signifying that, whatever the smoothing level up to 200, this very location remains a peak in the profile. On the other hand, arches are associated with both remaining spikes. However, they are out of proportion. Both are approximately one-meter high but the second spike is shaped by eight soundings and the third one is made by three soundings. We would expect the second arch to be roughly of the same height as the third one, although the latter is very much bigger.

In Figure 3.10, spikes located at 350 meters and 370 meters (this location features a set of three spikes) are source of some inconsistency as well. The big arch associated with the first spike is visible on the bottom graph. Small arches corresponding to the second set are more readily distinguishable on the zoomed fingerprint on the top graph. Whether the big arch constitutes an overestimation – or the small arches constitute an underestimation – of the

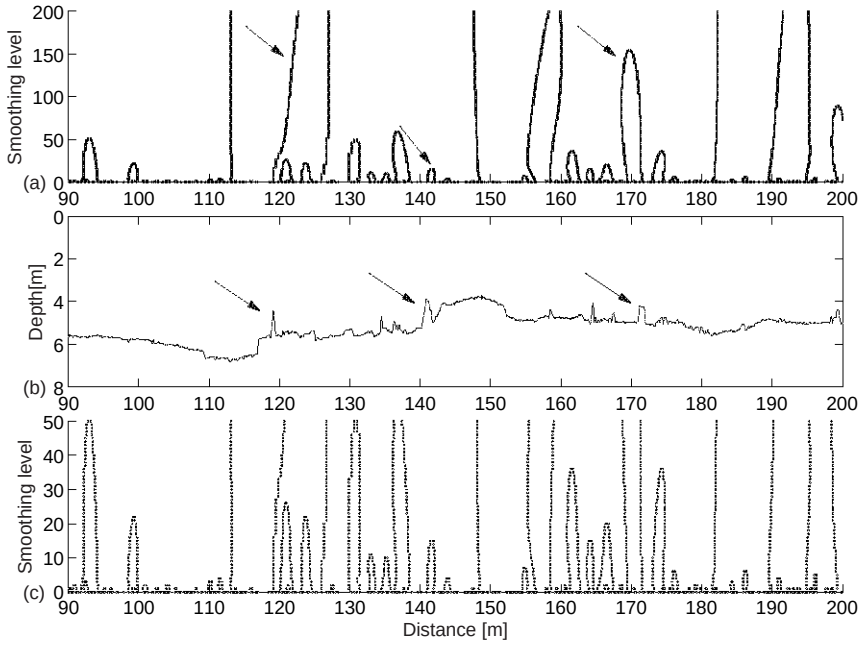

Figure 3.9: Fingerprint of bathymetry section presented in Figure 3.1a.

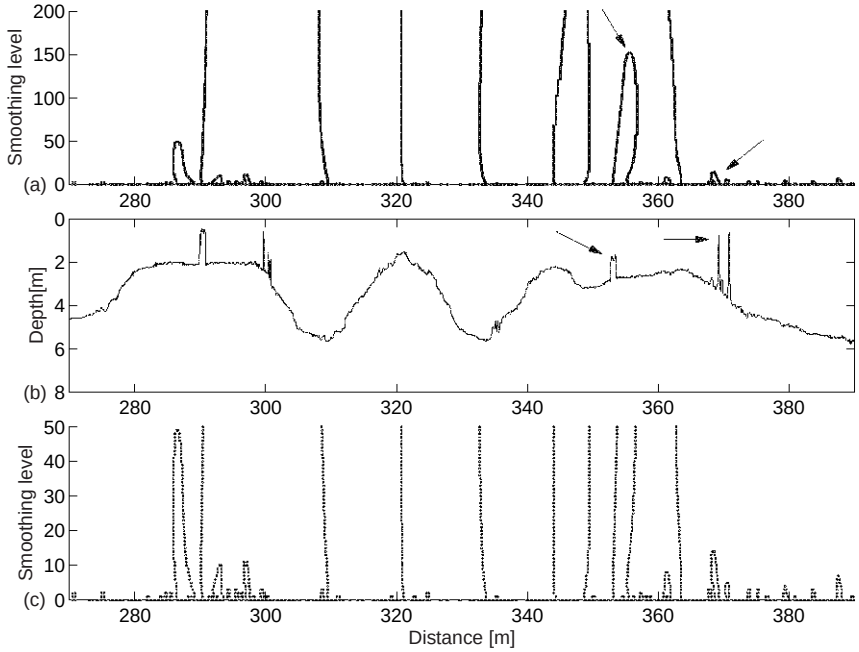

Figure 3.10: Fingerprint of bathymetry section presented in Figure 3.1b.

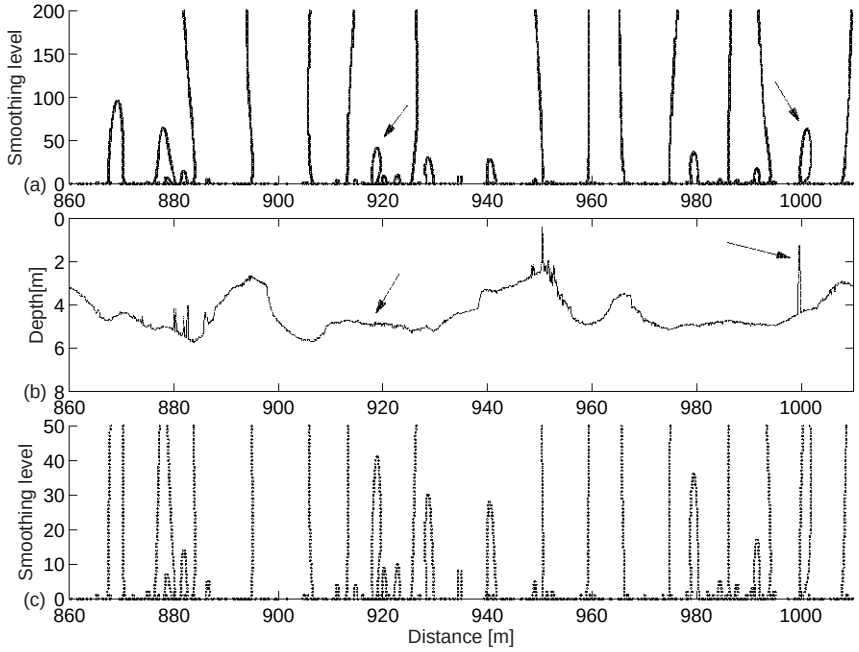

Figure 3.11: Fingerprint of bathymetry section presented in Figure 3.1c.

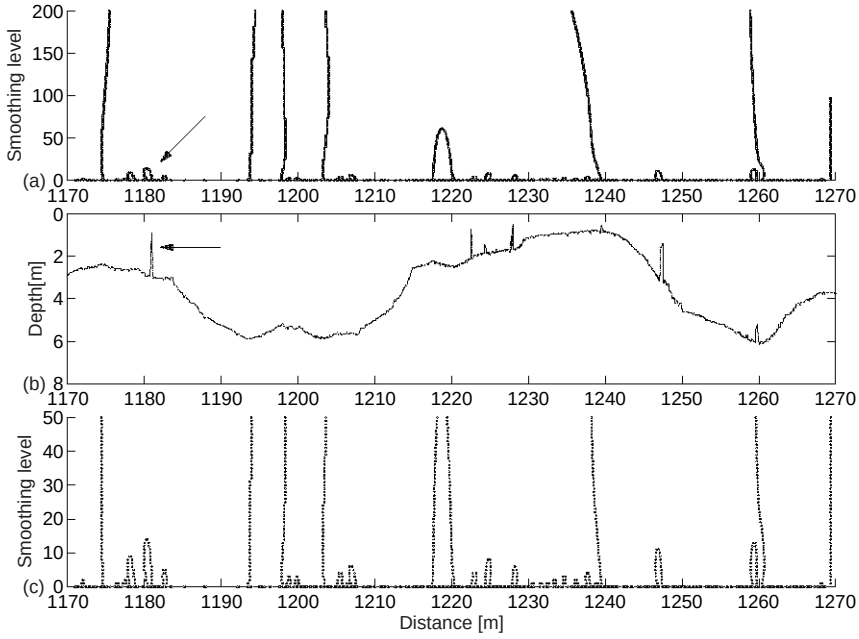

Figure 3.12: Fingerprint of bathymetry section presented in Figure 3.1d.

effect of related spikes is not certain. However, the discrepancy between those rises doubts again as to the reliability of this method to systematically pinpoint spikes and produce relevant information. Although spikes located at 370 meters are higher – approximately two meters –, they lead to disproportionate arch sizes.

In Figure 3.11, two arches (indicated by arrows) draw our attention. Both are similar in shape and size. Nevertheless, while the second one is associated with a three meter high spike, no spike is the source of the first arch.

Finally, the fingerprint in Figure 3.12 is more coherent by itself – spikes of similar size lead to equivalent arches –, yet fails to provide us with a consistent representation when compared with previous fingerprints. The spike indicated by an arrow in Figure 3.12 is approximately two meters high with an arch closing at the 15^{th} level of smoothing. Nonetheless, the second exhibited spike in Figure 3.11 roughly spans three meters while its associated arch persists for more than 50 levels of smoothing (it closes around the 60^{th} level).

The fingerprint method, as it was implemented and with the sampling interval that we have used, does not produce consistent, valuable information. Although it is not clear as yet how the technique could be improved, it should not be disposed of too rapidly. We ought to keep in mind that our framework departs from Bergeron's in two major ways, which is very likely the cause of the lack of success in our situation. First, Bergeron (1996) worked on coarse gravel-bed streams while the Sulphur and Guadalupe rivers have a sandy bed. Secondly – and probably the most influenceable factor –, our mean sampling interval is of an order of magnitude larger than Bergeron's. Although fine enough for gathering the general bathyemtry structure, it might be inadequately coarse when scale-space analysis is to be employed. Such sampling interval is able to catch spikes – and render them as erratic noise in the signal – but is *not* appropriate to seize the structure of the spike, caused by plausible piece of woody debris. It it questionable that such erratic spikes as those shaped by a single sounding should actually contribute the bed structure. Another reason for that conjecture is that most of the severe spikes have a relative height of more than one meter. Hence, it is extremely unlikely that the causing woody debris be lying on the stream bed *at the surveying location*. Consequently, they do not belong to the stream bottom structure per se. It should not be too unexpected that a technique devised to analyzing bed structure fail to represent these particular pieces of LWD.

3.5 Final comments on statistical techniques

It was attempted to make use of statistical analysis techniques in order to gather information on severe spikes featured within finely-sampled bathymetry profile. The most likely cause of these spikes is the presence of LWD. Among the experimented methods were basic ones (moving average, standard deviation) as well as other inordinary methods (semivariogram, scale-space analysis). It should be pointed out that none of these methods proposes a systematic way – that is, a procedure that would not require human interaction or interpretation – of identifying spikes and outputing their locations.

However, simpler techniques turn out to produce more relevant information than sophisticated ones. The former were essentially useful to ensuring the presence of spikes by combining the use of moving averages (eventually indispensable for any modeling step) and standard deviations. The sole use of moving averages inevitably conducts to a loss of information, namely disappearance of spikes – but not all of them. The latter property renders it useless as an identification tool. A moving average is mereley a low-pass filter of poor quality and only ensure narrow spikes (higher-frequency components) to be filtered out effectiveley while preserving wider spikes by smoothing them out. Any utilization of such smoothened data in modeling would produce unrealistic bathymetry profiles, caused by the presence of spurious flattened peaks that do not naturally belong to the stream bed. This observation suggests that an effective filter be found. This issue is the current reasearch topic of the authors. Superimposing the standard deviation onto the moving average permits to manifest the presence of spikes where the profile has been flattened out. Nevertheless, it also turns out that broader spikes generally engender a small standard deviation. It then becomes arguable whether the smoothened peaky feature is part of the bed profile or comes from actual LWD.

Among the more sophisticated methods, the semivariogram is designed to account for the general pattern of bathymetry profiles. When using very fine sampling, the semivariogram has been proved an effective tool allowing the separation of stream-bed roughness into a grain and a form component – for the case of gravel beds. As for sand-bed profiles, the semivariogram usually gives rise to relevant parameters associated with sand ripples. The profile that we obtained does not present the characteristics of a gravel-bed profile, nor that of a sand-bed profile. The most likely cause of this result is the use of a inappropriately coarse sampling interval, a factor a 20 larger than that used by Robert (1988).

Although scale-space analysis seems to be an effectual way of rendering a two-dimensional picture of the general structure of bed profiles, our attempt to obtain a representation of spurious spikes was pervaded with inconsistencies. As mentioned above, it is not certain whether a finer sampling interval would have led to more coherent fingerprints. It is very conceivable that spurious spikes act less as bathymetry structural components and consist more of noise that should be filtered out. Such interpretation makes these iconsistencies less surprising, as these alleged pieces of woody debris are not part of the general bathymetry structure. These must therefore be filtered out before any further attempt of analyzing the bed structure. Spectral or wavelet analysis should provide a way of doing so and presumably constitutes the direction to follow.

Chapter 4

Filtering techniques

The distortion of the averaged bathymetry (see Figure 2.3) – leading to erroneous interpolated grid data – requires that spurious spikes be removed. A first attempt of identifying the latter had been presented in the previous chapter. Although a combination of simple statistical methods (such as moving average and standard deviation) proved to be relevant in pointing out the presence of spikes, it does not provide a systematic way of dtermining the spike locations. Although effectual in rendering two-dimensional pictures of bed profile structures (Bergeron, 1996), scale-space analysis did not fulfill our expectations either. As previously suggested, it is very conceivable that spurious spikes act less as bathymetry structural components and consist more of erratic noise that should be filtered out.

Even though the sampling interval is fine enough for gathering the general bathyemtry structure, it might be inadequately coarse when scale-space analysis is to be employed. Such sampling interval is able to catch severe disruptions of bathymetry profile – and render them as erratic noise in the signal – but might *not* be appropriate to seize the structure of these very disruptions, caused by plausible piece of woody debris. It it questionable that such erratic spikes as those shaped by a single sounding should actually contribute the bed structure. Another reason for that conjecture is that most of the severe spikes have a relative height of more than one meter. Hence, it is extremely unlikely that the causing woody debris be lying on the stream bed *at the surveying location*. Consequently, they do not belong to the stream bottom structure per se. It should not be too unexpected that a technique devised to analyzing bed structure fail to represent these particular pieces of LWD. Under the scope of this assumption, inconsistencies pervading the 2D picture of bed structure are less surprising and more legitimate.

The aforementioned considerations suggest that, prior to analyzing river bottom structure, those extraneous spikes need be removed. Because spikes present these sharp depth transitions in the signal (depth changes of up to three meters from one sample to the next), they are associated with high-frequency components in the spectral domain. Similarly, for time-dependent signals, high-frequency components are produced by sharp changes of signal values within a short time period. By its broad objective of modifying certain frequencies relative to others, filtering naturally comes to mind (Oppenheim and Schafer, 1999). The following subsections present linear and nonlinear filtering techniques, all designed to remove high-frequency components in a signal. In order to compare filters, an artificial bathymetry has been synthesized (see Figure 4.1).

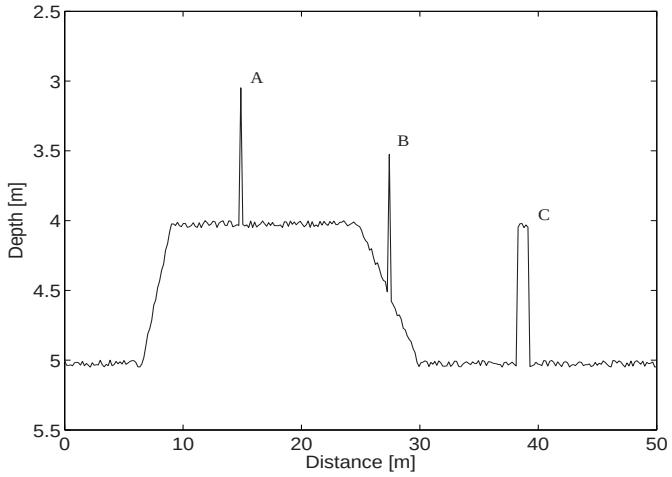

Figure 4.1: Synthesized bathymetry providing a benchmark for filters. Both narrow spikes are made up of one data point while the broader spike contains six data points.

Synthesizing an artificial bathymetry allows for including bottom features without any restrictions. Hence, it gives the possibility to add sharp edges or severe peaks – we intend to remove the latter – that are known to constitute difficulties to certain techniques. Assessment of what it takes for a particular technique to fail is also easily rendered possible. Nevertheless, one should keep in mind that a manufactured bathymetry will never succeed at faithfully representing natural landforms. There is therefore no guarantee that a successful method for artificial profiles yield similar-quality results for realistic profiles. Our synthesized bathymetry features two sharp edges as well as three spikes, two narrow (labels A and B) and one broader (label C) in Figure 4.1. The left edge is particularly steep and is not very likely to be encountered in reality. The right edge is not as steep (the slope is similar to that of Figure 2.5) but the spike lying on the slope (label B) *a priori* constitutes an extra difficulty. Therefore, a challenge for a filter is to be able to preserve sharp edges while filtering out spikes. A trade-off is not acceptable.

Noise has been added to the profile – after peaks were created – to liken small-scale erratic components that we may identify on natural bathymetric profiles (e.g., see Figure 2.5 and Figure 2.7). In order to quantitatively evaluate off-trend components, a closer look at the Guadalupe River data was deemed necessary. We focused on the bathymetry profile between 10 and 25 meters (see Figure 2.7) and lowpass-filtered it to obtain the trend. This step is depicted on Figure 4.2a. The trend was then subtracted from the original bathymetry, allowing on Figure 4.2b for a direct assessment of what the noise should be in the artificial profile. A peak-to-peak noise of 0.05 meter has been added to the synthetic bathymetry. The noise was modeled with a random number generator of uniform distribution between 0 and 1. Each random number was then multiplied by 0.05. The bathymetry data off the trend are believed to be caused by a combination of several factors. Those are most likely real bathymetric features but could also be boat pitching and rolling while recording depth soundings. Finally, inherent measurements errors should not be ruled out, although their influence should be very small.

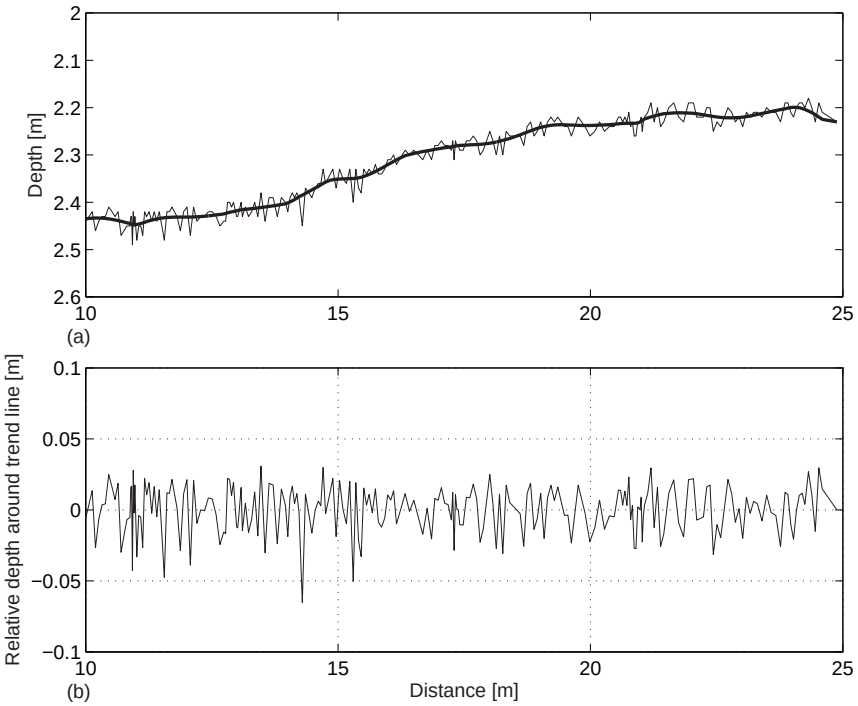

Figure 4.2: Both graphs focus on the Guadalupe River bathymetry between 10 and 25 meters (see Figure 2.7 for the entire profile). (a) The thin line is the original bathymetry and the thick line is a lowpass filtered version (cutoff normalized frequency of 0.1) showing the trend. (b) Difference between both bathymetries gives insight into the erratic component of the measurement.

4.1 Linear filtering

The advantage of utilizing *linear* filtering is dual: mathematical analysis is relatively simple and execution is very fast. Both those characteristics are closely associated with the convolution nature of linear filtering. Once a filter has been designed, a finite-length kernel must be convoluted with the signal. A filter kernel is a truncated version in the time domain of the inverse Fourier Transform of the ideal filter in the frequency domain. Linear convolution can efficiently be computed using Fast Fourier Transform (FFT) algorithms, via circular convolution. Basic FFT algorithms can perform this operation in $\mathcal{O}(L \log_2 L)$, where L is the size of the signal (Oppenheim and Schafer, 1999). However, linear filters compute new values from neighboring data points; they tend to blur the signal and remove sharp details that do not necessarily consist of noise.

Linear filters fall into two categories: Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters. FIR filters are unconditionally stable – no feedback is used – but require larger convolution kernels than IIR filters, rendering their execution slower. Nevertheless, the latter are only conditionally stable – feedback is used – and particular care must be taken when designing them (Oppenheim and Schafer, 1999). In the following, the results of an FIR filter of order 50 and an IIR filter of order 10 are presented.

As explained above, extraneous spikes – that we want to cut out – inherently present sharp depth changes from one sample to the next. High-frequency components are therefore associated with them in the spectral domain. Among digital filters, lowpass filters are designed to remove all signal components whose frequency is beyond the so-called cutoff frequency while all low-frequency components are preserved (in the case of ideal lowpass filters). Since spikes lead to high frequencies, the use of a proper lowpass filter is expected to fulfill our goal.

Whatever lowpass filter is employed, a cutoff frequency must be appropriately chosen, hence constituting a major drawback of linear filtering. Indeed, the frequency content of the signal is unknown *a priori* so that a cutoff frequency cannot safely be selected prior to examining the Fourier transform of the signal, which yields its frequency content. Even so, the relevance of such frequency content might be questionable: higher-frequency components are associated with sharp transitions, be it a spike or an edge. Cutting off these components will inevitably lead to edge blurring, a phenomenon that we wish to avoid. Furthermore, broader spikes contain lower-frequency components that will be conserved with a lowpass filter. This occurs because, among samples shaping the spike, those at the middle have values close to that of adjacent samples. Those middle samples do not lead to high frequencies and might not be effectively filtered out. The selection of an adequate cutoff frequency is far from being straighforward.

4.1.1 FIR filter

A Hamming-windowed linear filter of order 50 (with four different cutoff frequencies) has been used as a first experiment to filter out spurious spikes (see Figure 4.3). Windowing is used to truncate the ideal lowpass filter (that would be of infinite duration) in the time domain.

It is evident from Figure 4.3 that there is a trade-off between the goals that we intend to achieve: preserving edges while cutting out spikes. Whereas the use of a relatively low cutoff frequency leads to a fair removal of the three spikes, both edges a unacceptably smoothened out, which distorts the original profile (e.g., Figure 4.3a). The opposite phenonemon occurs with the highest cutoff frequency. Edges are very well preserved but the higher cutoff frequency somewhat conserves spikes too. Finally, it should be pointed out that none of the cutoff frequencies yields a satisfactory removal of the broader spike (label C). A bump still lingers.

4.1.2 IIR filter

The most common IIR filters are the Butterworth, Type 1 Chebyshev, Type 2 Chebyshev and Elliptic filters. Although all but Type 2 Chebyshev have been tested, only the results of Butterworth filtering will be presented. The other filters essentially give similar results when applied to our artificial bathymetry. A Butterworth filter of order 10 with the same four cutoff frequencies has been employed (see Figure 4.4).

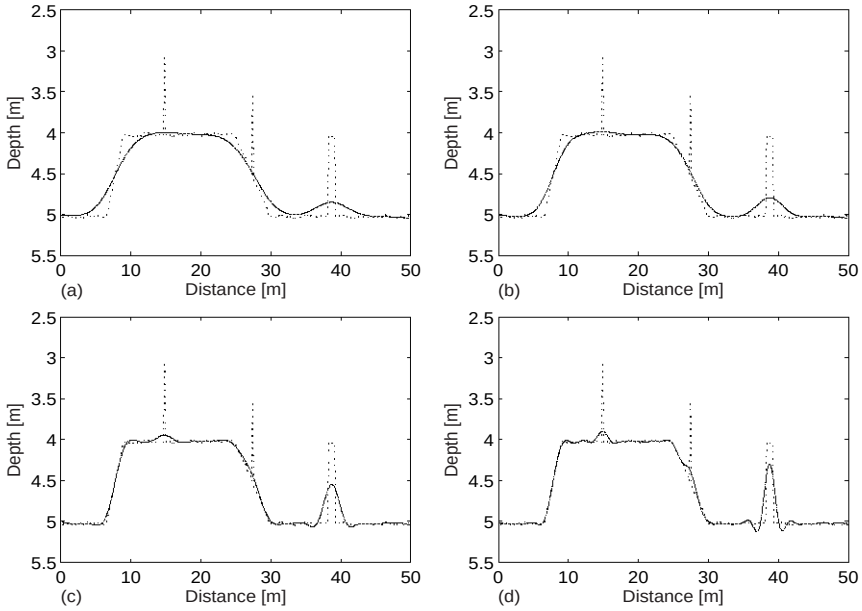

Figure 4.3: The dotted and solid lines represent the original and filtered signals, respectively. An FIR filter of order 50 has been used. Normalized cutoff frequencies are: (a) 0.025, (b) 0.05, (c) 0.1 and (d) 0.15.

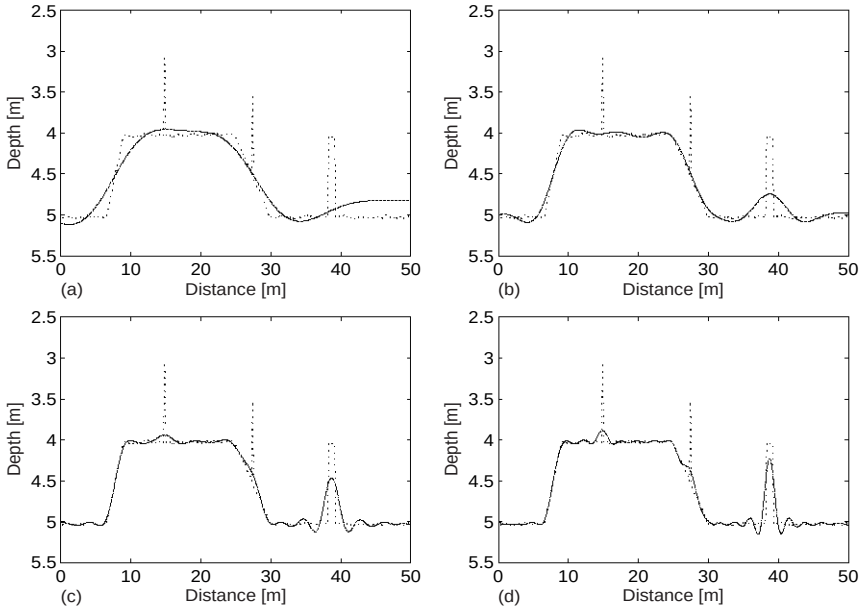

Figure 4.4: The dotted and solid lines represent the original and filtered signals, respectively. A Butterworth filter of order 10 has been used. Normalized cutoff frequencies are: (a) 0.025, (b) 0.05, (c) 0.1 and (d) 0.15.

Overall, Butterworth filter performance is less satisfactory than that of FIR filtering. In addition, an attempt of increasing its order leads to instabilities for the lowest cutoff frequency so that 10 is the maximum allowable order with this set of frequencies. The presence of ripples around the left-hand broader spike (see Figs 4.4c and 4.4d) renders the filtered bathymetry even less faithful to the original profile devoid of spikes.

4.2 Nonlinear filtering

As illustrated in Figs 4.3 and 4.4, linear filtering trades off between spike removal and edge preservation. Moreover, it cannot successfully dispose of broad spikes (label C), which contain both low- and high-frequency components.

Spikes added to our artificial bathymetry (and featured in Figure 2.5) are typical of so-called *impulse noise*, namely large, isolated "errors". Some nonlinear techniques, however, perform very well at accomplishing this tandem objective of removing impulse noise and keeping edges undisturbed. Unfortunately, their execution time is slower than that of linear filtering techniques – sorting operation is required for each data point – and mathematical analysis is generally more complex. Two nonlinear filters are introduced: the erosion filter and the median filter. The latter is widely used in image processing to filter out "salt-and-pepper noise", that is bits that are flipped with a certain probability. Median filtering is generally capable of restoring the original image without blurring the edges.

Both techniques use the concept of order statistics. For each data point $\tilde{x}$ in the sequence $x(n)$, a length-$(2N+1)$ set of neighboring data points is formed by taking those N points preceding $\tilde{x}$ and those N points succeeding it. This set is then algebraically ordered. Erosion filtering works by replacing $\tilde{x}$ with the minimum of the ordered set. Median filtering is obtained by replacing $\tilde{x}$ with the median of the ordered set. These steps are depicted in Figure 4.5 for median filtering with $N = 2$, where $x(n)$ is the original sequence containing a spike and $y(n)$ is the filtered sequence. Graphs (a) and (c) show filtering of sample 7. The order statistics centered at this sample (see dashed box in Figure 4.5a) contains the following points {2.5 3.0 6.5 8.0 4.5}. The ordered set is {2.5 3.0 4.5 6.5 8.0} so that the median is 4.5. the last step consists in replacing sample 7 by this value. This is illustrated in Figure 4.5c where the circled sample is the result of the filer. Filtering sample 8 follows the same process and is shown on graphs (b) and (d). The order statistics is {3.0 6.5 8.0 4.5 5.0} and the ordered set becomes {3.0 4.5 5.0 6.5 8.0}. The median is 5.0, which becomes the value of substitution in the filtered sequence (circled sample on graph (d)). Note that erosion filtering takes the minimum of the order statistics instead of the median.

Filtering the first and last N data points of the signal is problematic in that too few elements are available in the ordered set. Common practice is to append the signal with end values: N new data points (each equal to the first data point of the signal) are appended to the beginning and N new data points (each equal to the last data point of the signal) are appended to the end.

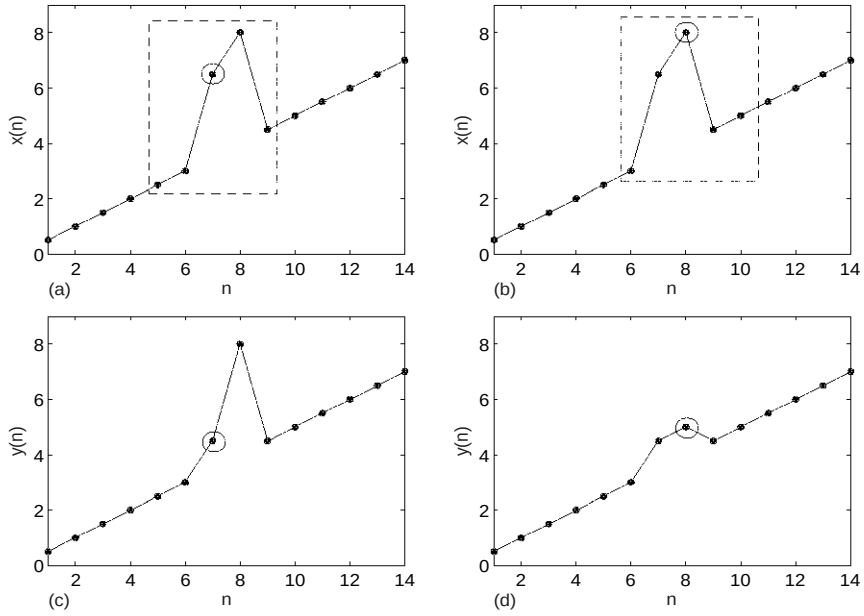

Figure 4.5: Illustration of median filtering for $N = 2$. $x(n)$ and $y(n)$ are the original and filtered sequences, respectively. The dashed box contains the five samples to be algebraically ordered. Plots (a) and (c) show the filtering of sample 7 and plots (b) and (d) show the filtering of sample 8. The sample of interest is circled in each case.

4.2.1 Erosion filter

By definition, erosion filtering works by taking the minimum of any ordered set. A modification is necessary in our case. The bathymetry is defined by the depth rather than height relative to an arbitrary datum. Thus, erosion filtering was implemented by taking the maximum instead of the minimum of each ordered set; the spikes consisting of lowest values. The effect of erosion filtering is shown in Figure 4.6, where four different filter lengths have been used.

Whereas the length-3 erosion filter is able to eradicate both impulses (see Figure 4.6a), it cannot remove the broader one. The only effect of the filter is to erode the spike by two data points (from six to four data points). A length-7 erosion filter is required to remove the broad spike (see Figure 4.6b). Nevertheless, with higher-order erosion filters, undesirable side effects appear. They keep eradicating all spikes but also erode the bump, thereby distorting the bathymetry and generating a discrepancy with the original, devoid of spikes, bathymetry. Moreover, the higher the order, the larger the discrepancy (e.g., Figure 4.6d). Depending whether a surveyed bathymetry contains broad spikes, a high-order filter might be necessary – at the cost of distorting other bottom structural elements – which is unwelcome.

4.2.2 Median filter

Median filtering has been experimented with four filter lengths. Results are shown in Figure 4.7.

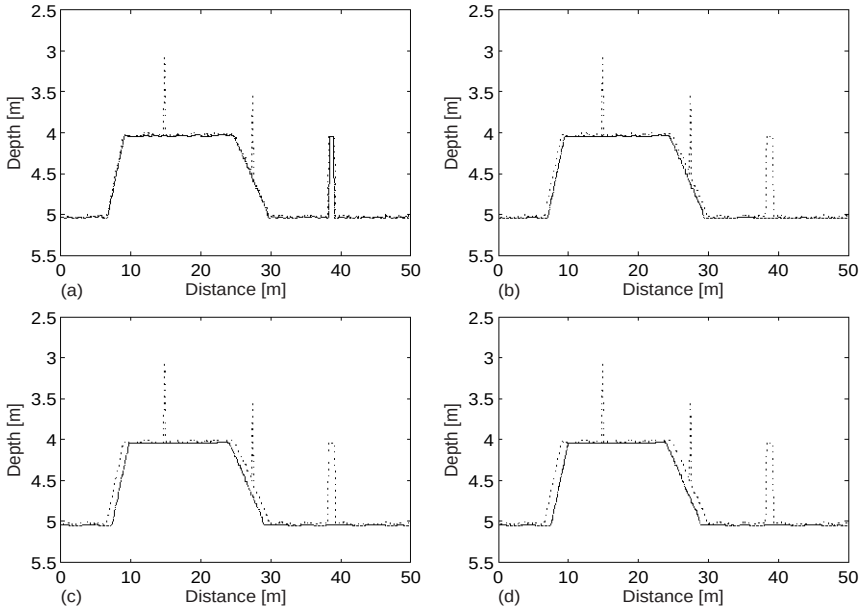

Figure 4.6: The dotted and solid lines represent the original and erosion-filtered signals, respectively. The following filter lengths have been used: (a) 3 ($N = 1$), (b) 7 ($N = 3$), (c) 11 ($N = 5$) and (d) 13 ($N = 6$).

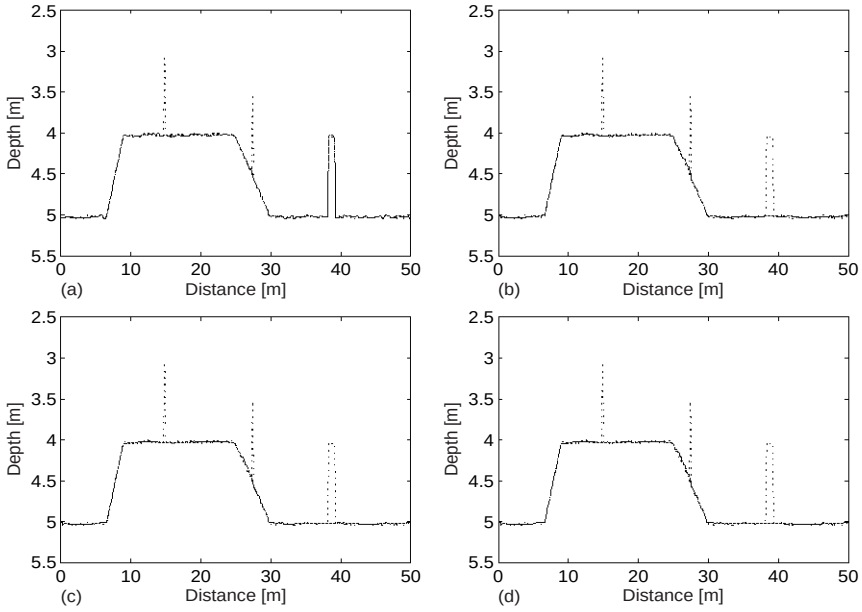

Figure 4.7: The dotted and solid lines represent the original and median-filtered signals, respectively. The following filter lengths have been used: (a) 3 ($N = 1$), (b) 13 ($N = 6$), (c) 17 ($N = 8$) and (d) 25 ($N = 12$).

Unlike erosion filtering, a length-13 median filter is required to eradicate the broader spike (see Figure 4.7b) while length-3 filters already remove impulses, as shown in Figure 4.7a. Two higher-order median filters (length-17 and length-25) have been used in order to show how both bump edges are left undisturbed while all spikes are perfectly eradicated (see Figs 4.7c and 4.7d). In addition to the nonlinearity of median filtering, the only price to pay for such satisfactory results is the use of a relatively high order – hence, increased computational effort – if broad spikes characterize surveyed bathymetries. Nevertheless, not only do we do away with the trade-off between edge rentention and spike rejection (inherent to linear techniques), the filtering computational complexity also remains linear in $\mathcal{O}(kL)$, where L is the cardinal of the data set to be filtered and k is the complexity associated with finding the median of the set at each data point. The best "find median" algorithm – developped by Hoare (1961) – has a performance $k = \mathcal{O}(2N + 1)$ and is therefore linear in the filter order. Nevertheless, for reasonable data set sizes, linear filtering execution – with performance $\mathcal{O}(L \log_2 L)$ – remains faster. For instance, a data set should contain over 10^7 data points for the length-25 median filter performance to overcome that of linear filtering.

4.3 Applications

It is evident from the benchmark that linear filtering is largely outperformed by nonlinear filtering, and median filtering in particular. Although execution of the latter is slower, this drawback is largely compensated by its effective capacity for eradicating spikes of various widths. Furthermore, under the assumption that the spikes be mere artifacts, a method consisting in replacing data points belonging to spikes by data points belonging to actual stream bottom – i.e. erosion and median filtering – is physically sounder than any linear filtering averaging technique.

Both nonlinear filters have been utilized on three 100-meter sections of surveyed bathymetry. Median-filtered bathymetry is shown in Figure 4.8 while erosion-filtered bathymetry is depicted in Figure 4.9.

Although the erosion filter length required to remove all spikes is almost half that of median filtering, side effects are unacceptable. This latter conclusion is already visible on Figure 4.9 but direct comparison of both filters makes it evident. In Figure 4.10b, the asbolute difference between original and filtered bathymetries is shown for both filters. An assessment of filter performance – as regards spikes removal – can be made by examining Figure 4.10a. For instance, the interval between 320 and 330 meters is expected to be undisturbed since no severe spike is present. However, it is made of a somewhat sharp edge and thus constitutes an *a priori* difficulty for erosion filtering. Median filtering very satisfactorily conserves the bathymetry, yielding absolute differences that do not exceed 15 cm, that is a relative difference of 8 % at this spot. On the other hand, erosion-filtered bathymetry differs from the original one by up to 80 cm, which amounts to a 20 % relative difference at this spot. Overall, in accordance with our benchmark results regarding erosion filtering, the latter performs relatively poorly in presence of steeper edges (e.g., Figure 4.10 between 270 and 290 m and 300 and 350 m.).

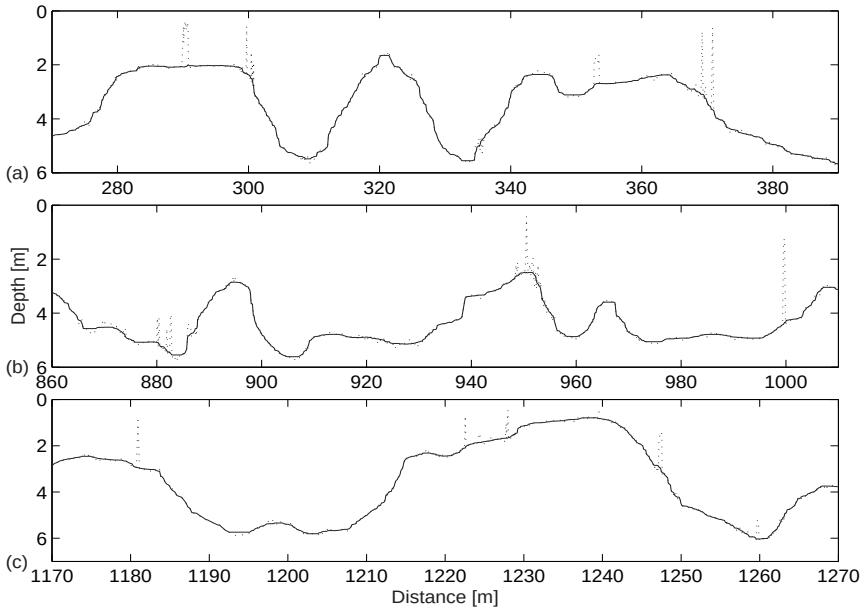

Figure 4.8: The dotted line and solid lines represent the original and filtered signals, respectively. A length-23 median filter has been used. This is the minimum length required to eradicate broad spikes situated at approximately 290 and 355 meters on graph (a).

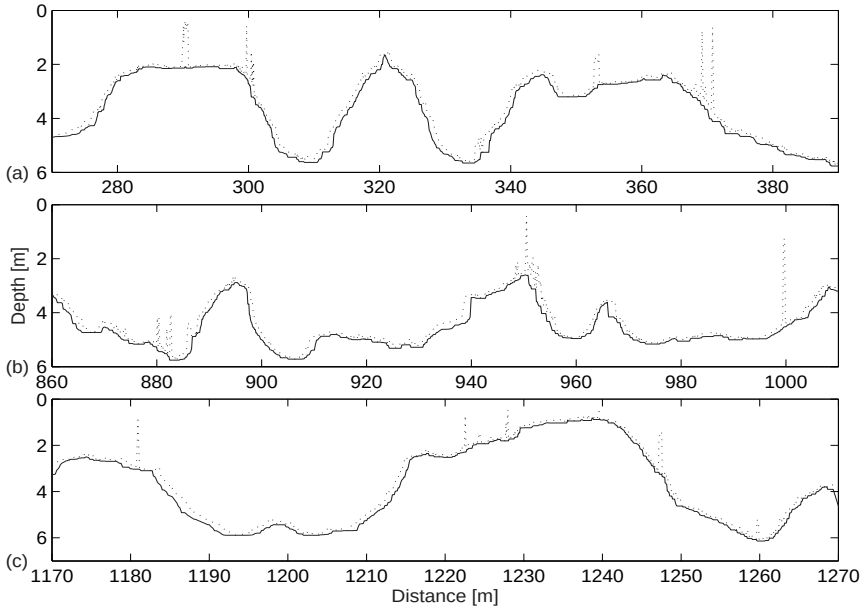

Figure 4.9: The dotted line and solid lines represent the original and filtered signals, respectively. A length-13 erosion filter has been used. This is the minimum length required to eradicate broad spikes situated at approximately 290 and 355 meters on graph (a).

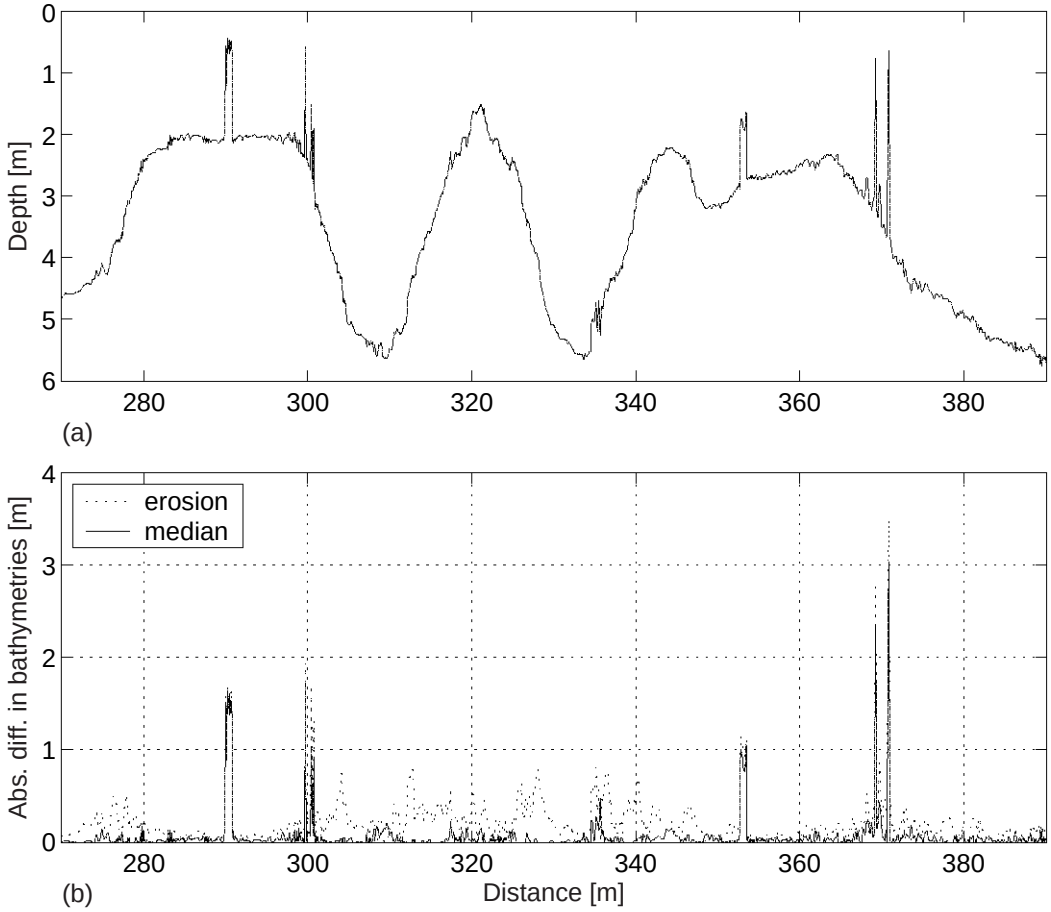

Figure 4.10: Evaluation of erosion and median filtering performance. (a) Original bathymetry. (b) Dotted and solid lines represent absolute differences between original and filtered bathymetries, for erosion and median filters, respectively.

Median filtering offers a very viable way of processing LWD-polluted bathymetry profiles. Unlike linear filtering, it does not encounter the inherent trade-off between spike rejection and edge retention so that sharp structural bed elements are preserved. In this respect, erosion filtering is of inferior quality as well in that it erodes large-scale bed structural elements, thereby distorting the bathymetry. Median filtering constitutes a practical way of disposing of spikes and producing a smooth, workable – e.g., for interpolation – bathymetry. It also consists of a helpful stream habitat analysis tool. Comparison of filtered and original bathymetries allows for identifying those data points whose value is beyond a fixed datum. It may then be decided upon whether these pinpointed data points belong to LWD, in which case a mapping of LWD locations within the stream is rendered possible (the technique has been employed in Osting *at al* (2003)) and is depicted in Figure 4.12.

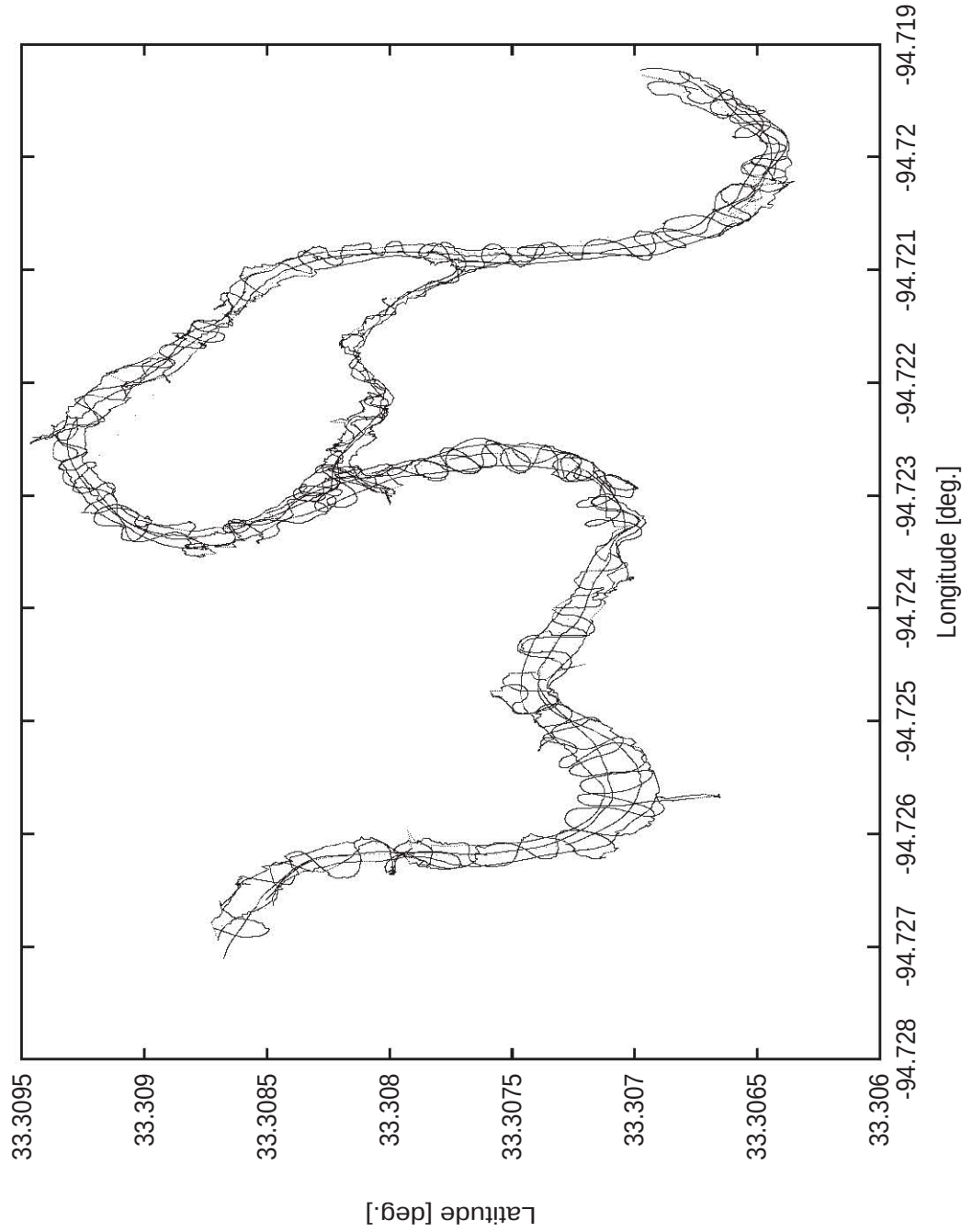

Figure 4.11: Scatter plot of all bathymetry survey tracks in Sulphur River

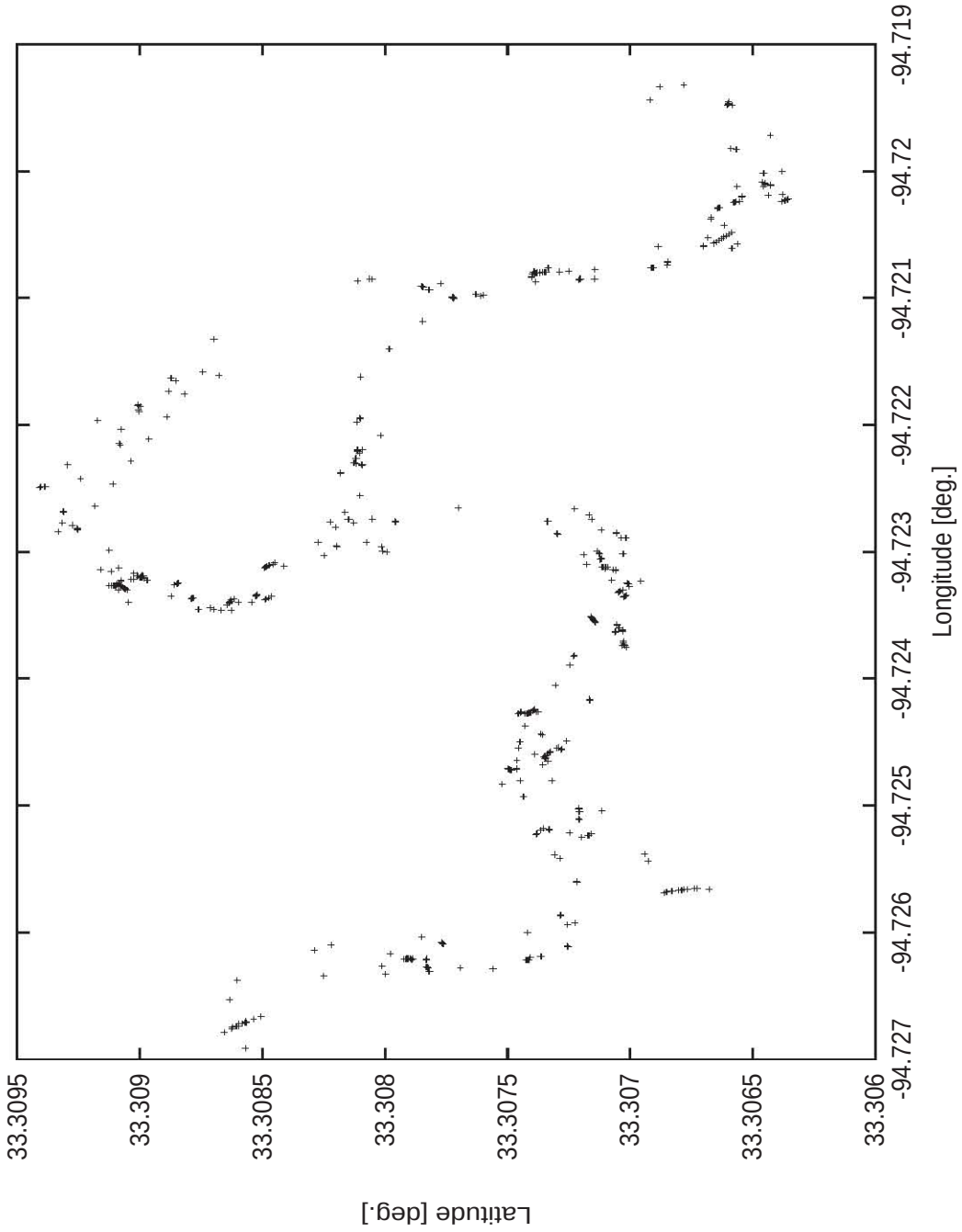

Figure 4.12: Mapping of LWD locations based upon spikes in the dataset that have a relative height greater than 0.5 m. The median filter is of length 23.

Chapter 5

Conclusion

The more frequent use of two-dimensional hydrodynamic rivers models also requires more detailed bathymetry surveys. For smooth bathymetries, there is little difficulty in developing accurate translations from survey data to model; however, in rivers with significant bottom structure – as is the case in the Sulphur River –, simple data averaging and interpolation methods may lead to misrepresentation of the bottom bathymetry.

Given the fine bathymetry gathered by Texas Water Development Board, it was necessary to identify in the data set what was true bathymetry from what was caused by large woody debris. To do so, the hypothesis was laid out that severe disruptive spikes in the data set be the signature of large woody debris. Two groups of methods have been investigated to serve our objective: statistical techniques and filtering techniques.

Simple statistical methods, such as moving average and standard deviation, were useful to diagnose the presence of spikes in the averaged set. However, spikes were inconsistently processed by the moving average: while narrow peaks were cut out, broader peaks were merely flattened and remained. None of these simple methods constitutes a practical, systematic way of identifying spikes. Turning to more involved techniques, such as semivariogram and scale-space analysis, did not provide us with a viable alternative.

Unlike the first group of methods, the second group focused more on separating spikes from the true bathymetry. An artificial bathymetry presenting both sharp edges and spikes was manufactured to assess the effectiveness of techniques in eradicating spikes while preserving edges. Linear lowpass filtering was first tried out but failed at fulfilling that goal. Nonlinear filters – median and erosion filters – were far more succesful at preserving sharp edges and cutting off spikes. Median filtering is specifically designed to serve that goal and is therefore not hindered by the inherent tradeoff associated with linear filters. Erosion filtering, while eradicating spikes, also erodes large-scale bathymetric features and is therefore outperformed by median filtering, which leaves large-scale features undisturbed.

Median filtering may be considered a very effective – and simple – technique to filter out spikes generated by the presence of submerged large woody debris in rivers. In addition, not only does median filtering yield a smooth, applicable bathymetry, it also gives LWD locations, which is of high interest in aquatic habitat analysis.

1D one-dimensional

2D two-dimensional

DTFT Discrete Time Fourier Transform

FIR Finite Impulse Response

IIR Infinite Impulse Response

LWD large woody debris

FFT Fast Fourier Transform

GPS Global Positioning System

TWDB Texas Water Development Board

Appendix A

Pictures of LWD in Sulphur River

This appendix contains more pictures of LWD in the Sulphur River. All pictures were taken on (date ?) by Texas Water Development Board (TWDB) during low flow.

Figure A.1: Emergent LWD in the Sulphur River (Northeast Texas).

Figure A.2: Emergent LWD in the Sulphur River (Northeast Texas).

Figure A.3: Emergent LWD in the Sulphur River (Northeast Texas).

Figure A.4: Emergent LWD in the Sulphur River (Northeast Texas).

Figure A.5: Emergent LWD in the Sulphur River (Northeast Texas).

Appendix B

Bathymetry Process 1.1: User's guide

B.1 Introduction

The program *bp* provides tools to process raw bathymetry data, export processed data for use under Matlab, identify LWD (using median filtering) and export filtered bathymetry.

For Linux users, if Gnuplot is available on the machine where *bp* is installed, plotting is an option and allows for producing scatter plots of depth measurement locations as well as original and filtered bathymetries. The latter is particularly convenient for assessing the quality of a filter before quitting *bp*.

The interface between C and Gnuplot is provided by the gnuplot_i library written by N. Devillard (ndevilla.free.fr).

B.2 Installation

This section describes the requirements and the steps necessary to compile and run *bp*.

B.2.1 Requirements

For **Linux**, the following is required :

- C compiler (gcc).
- make (to interpret the Makefile).
- Gnuplot is optional.

For **Windows**, a C or C++ compiler is required. I have not tried to compile and run it under Windows but I was told it works well.

The Windows version does not support Gnuplot. The interface gnuplot_i uses POSIX pipes that are not supported under Windows.

B.2.2 Compilation of the source

Under **Linux**, the only thing to do is to type `make` from within the directory containing the source. Note that `make all` or `make os_linux` will do the same.

Under **Windows**, the first thing to do is to edit the file `os.h` and comment out the line `# define OS_LINUX`. This will ensure that all Gnuplot-related instructions in the source code be removed by the preprocessor. It is now ready for compilation. By using the Makefile, just type `make os_win`. Without using the Makefile, I don't know but make sure not to include `graphs.*` and `gnuplot_i.*` in the compilation.

B.2.3 Completion

When installing under Linux, make sure the directories `gnuplotdata` and `gnuplotfigs` are in the same directory as the binary.

B.3 Utilization

bp must be invoked with two arguments. The usage is

`bp` FILE PREFIX

where FILE is the file containing the raw data and PREFIX is the prefix of all output files (more on that below). Note that invoking

`bp --help`

will display a short help message reminding the user how to run *bp*.

B.3.1 Processing raw data

This corresponds to item 1 in the MAIN MENU and leads to the RAW DATA MENU that permits the user to choose among three data file configurations. The menu items are self-explanatory.

Processing raw data must be performed prior to any other task. It reads the file, stores all data into memory and computes statistics (such as mean depth per GPS position).

B.3.2 Identifying Large Woody Debris

This selection corresponds to item 2 in the MAIN MENU and leads to the LWD IDENTIFICATION MENU from which two identification methods are proposed. Both are based on median filtering[1]. Under each selection, the user is required to enter a filter half-length.

[1]Refer to section B.4 for a brief overview of median filtering or section 4.2 for more detailed explanations.

First method The first method filters the data with a *single filter half-length*. The user is then invited to enter a discriminatory height h. LWD identification works as follows. The difference between original and filtered bathymetries is calculated and all spikes whose height is larger than h are viewed as belonging to a piece of woody debris. In parallel to this process, two files are created. Median-filtered data are exported into PREFIX`.flt` and LWD locations are exported into PREFIX`.lwd`. The format of PREFIX`.flt` is

```
LON1 LAT1 H1
LON2 LAT2 H2
...
```

where LON, LAT and H are the longitude, latitude and filtered bathymetry data point, respectively. The format of PREFIX`.lwd` is

```
LON1 LAT1 I1
LON2 LAT2 I2
...
```

where I is the data point index within the data set.

Second method The second method performs median filtering with several filter half-lengths. If K is the filter half-length entered by the user, filtering will be performed with filter half-lengths 1 2 ... K. For each filtering, the two same files as above are created, namely PREFIX_`##.flt` and PREFIX_`##.lwd` where `##` is replaced by one of the filter half-lengths. The second method allows for distinguishing between spike widths, as explained in the appendix.

B.3.3 Exporting processed data

Processed data may be exported into Matlab scripts. The following files are created when selecting item 3 in the MAIN MENU. Within each of these files, the first lines describe the content.

- PREFIX_`sum.m` : This file contains summarized data per GPS position.
- PREFIX_`fin.m` : This file contains all depth measurements with GPS positions linearly interpolated between known positions.
- PREFIX_`lum.m` : Obselete and not useful anymore. Will be disabled in subsequent versions.

B.3.4 Plotting

Plotting with Gnuplot is only available in the Linux version[2]. Selecting item 4 in the MAIN MENU enables the PLOTTING MENU. Three graphs may be produced, either in paper version (Encapsulated Postscript format) or on the screen. The first two items in this menu produce scatter plots of GPS locations. The 'Fine scatter plot' option linearly interpolates between

[2]Who wants to use Windows anyway ?

GPS positions to plot all depth measurement locations. The third menu item produces a two-graph page comparing original and filtered bathymetries. For each of these plots, the user is asked whether to produce a paper graph or not. In addition, for the third graph, the x-axis range must be entered (or enter 0 0 for the entire range).

B.4 Median filtering

A median filter of length $2N + 1$[3] works the following way : for each depth sounding, we build a data set made of the N soundings preceding it, the N soundings following it and the sounding itself. We now have a data set of $2N + 1$ soundings. The center sounding is replaced by the median of this set. This type of filtering ensures that all edges be conserved. Therefore, sharp changes in the bathymetry will not be affected. Only spikes will be totally removed. Specifying a filter half-length of N will remove spikes whose width contains up to N soundings.

[3]Note that the user enters the filter half-length N.

Appendix C

Code listing

```
/* ----------------------------------
 *
 *      main.c
 *
 *      Laurent White
 *
 *      Date of creation : 2002-12-19
 *
 * ---------------------------------- */

# include <stdio.h>
# include <stdlib.h>
# include <string.h>

# include "os.h"
# include "preprocess.h"
# include "raw.h"
# include "identifylwd.h"
# include "stdev.h"
# include "postprocess.h"

/* If compiling under Linux, include Gnuplot extension */
# ifdef OS_LINUX
   # include "graphs.h"
   # include "gnuplot_i.h"
# endif

int main ( int argc, char **argv )
{
   int n_gps_positions; /* Number of different locations
            * found in input file */
   gps_position *gps_positions;  /* Array of struct gps_position */
   int main_menu_choice;
   int return_value;

   #ifdef OS_LINUX
   gnuplot_ctrl *h;
   #endif

   /* Handle arguments */
   if ( handle_arguments ( argc, argv ) == 0 )
      return 0;

   /* Allocate memory */
   gps_positions = (gps_position *) malloc ( MAX_GPS_POSITIONS * sizeof(gps_position) );

   /* Display signature */
   (void) display_signature ();

   /* Enter main menu loop */
   main_menu_choice = MM_NONE;
   n_gps_positions = -1;

   /* Initialization of Gnuplot */
   # ifdef OS_LINUX
   h = gnuplot_init ();
   # endif

   while ( 1 )
   {
      switch ( main_menu_choice )
      {
         case MM_RAW:
            /* Read raw data and compute statistics */
            n_gps_positions = process_raw ( argv[1], gps_positions );

            if ( n_gps_positions == 0 )   /* An error occurred in 'process_raw' */
            {
               free ( gps_positions );
               return 0;
```

```
            }

            if ( n_gps_positions == -1 )
               break;

            break;

         case MM_LWD:
            /* LWD identification */
            return_value = identify_LWD ( argv[2], gps_positions, n_gps_positions );

            if ( return_value == 0 )   /* An error occurred in 'identify_LWD' */
            {
               free ( gps_positions );
               return 0;
            }

            if ( return_value == -1 )
               break;


            break;

         case MM_EXP:
            /* Export to Matlab */
            return_value = export_to_matlab ( argv[2], gps_positions, n_gps_positions );

            break;

         # ifdef OS_LINUX
         case MM_GRA:
            /* Plotting */
            return_value = plotting_menu ( gps_positions, n_gps_positions, h );
            break;
         # endif

         case MM_EXIT:
            printf ( "Bye.\n" );
            /* Free memory */
            free ( gps_positions );
            return 1;
            break;

      }
      (void) display_main_menu ();
      printf ( "    Selection : " );
      scanf ( "%d", &main_menu_choice );

   } /* End main menu loop */

   # ifdef OS_LINUX
   gnuplot_close (h);
   # endif

   return 1;
}
```

```
/* ---------------------------------
 *
 *      os.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-02-21
 *
 * --------------------------------- */

# ifndef OS_H
# define OS_H

/* If compiling for Windows, comment the next line */
# define OS_LINUX

# endif /* OS_H */
```

```
/* ----------------------------------
 *
 *      defs.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-02-21
 *
 * ---------------------------------- */

# ifndef DEFS_H
# define DEFS_H

# define MAX_FNAME_SIZE 128

/* Define the reference locations */
/* For Raw data 1 */
# define LONGITUDE_1   94.0
# define LATITUDE_1    33.0
# define LONG_C_1      'W'
# define LAT_C_1       'N'

/* For Raw data 2 */
# define LONGITUDE_2   94.0
# define LATITUDE_2    33.0
# define LONG_C_2      'W'
# define LAT_C_2       'N'

/* For Raw data 3 */
# define LONGITUDE_3   97.0
# define LATITUDE_3    29.0
# define LONG_C_3      'W'
# define LAT_C_3       'N'

/* Maximum allowable number of distinct GPS positions */
# define MAX_GPS_POSITIONS 65536

/* Maximum allowable number of soundings at one GPS position */
# define MAX_SOUNDINGS 256

/* Conversion factor : number of feet per meter */
# define CONV_FEET_M 3.28083990

/* Earth radius */
# define RE 6400000

/* Structure defining a GPS position */
typedef struct
   {
      int     id;
      double   longitude;
      double   latitude;
      int   n_soundings;         /* Number of soundings */
      double   soundings[MAX_SOUNDINGS];  /* Depth soundings */
      double  timestamps[MAX_SOUNDINGS];  /* Time stamps of soundings */
      double   depth_mean;       /* Mean depth of soundings */
      double  depth_mean_smooth;    /* Depth of smoothed bath. */
      double   depth_var;        /* Variance of soundings */
      double  depth_var_smooth;
      double  slope;          /* Local slope (using the mean depths) */
      double   slope_mean;       /* Mean slope of all slopes between
                          this position and the next one */
      double   slope_var;        /* Variance of all slopes between
                          this position and the next one*/
      double   distance;         /* Distance between this position and
                            the next one */
   } gps_position;


/* Constants used in Main Menu management */

# define MM_NONE -1
# define MM_RAW   1
# define MM_LWD   2
# define MM_EXP   3
# define MM_GRA   4
# define MM_EXIT  9


# endif /* DEFS_H */
```

```
/* ---------------------------------
 *
 *      preprocess.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-02-19
 *
 * --------------------------------- */

# ifndef PREPROCESS_H
# define PREPROCESS_H

void display_help ();

void display_signature ();

void display_main_menu ();

int handle_arguments ( int argc, char **argv );

# endif /* PREPROCESS_H */
```

```
/* ---------------------------------
 *
 *      preprocess.c
 *
 *      Laurent White
 *
 *      Date of creation : 2003-02-19
 *
 * --------------------------------- */

# include <stdio.h>
# include <stdlib.h>
# include <string.h>

# include "os.h"
# include "preprocess.h"


void display_help ()
/*
   # Arguments : /
   #
   # Action : Display help.
   #
   # Return : /
*/
{
   printf ( "Usage : bp [FILE] [PREFIX]" );
   printf ( "\n\nThis program processes the raw data \
         contained in FILE and outputs all results\n" );
   printf ( "in files whose prefix is PREFIX.\n" );
   printf ( "Please refer to manual for detailed help.\n" );
   printf ( "\nFor bugs, please contact lwh@mail.utexas.edu (2003).\n");
}


void display_signature ()
/*
   # Arguments : /
   #
   # Action : Display signature.
   #
   # Return : /
*/
{
   int system_call;

   /* Execute 'clear' shell command */
   system_call = system ( "clear" );

   printf ( "\n" );
   printf ( "------------------------------------------------------------\n" );
   printf ( "This is bp (Bathymetry Process) version 1.1.              \n" );
   printf ( "Author : Laurent White (lwh@mail.utexas.edu).            \n" );
   printf ( "         University of Texas at Austin.                  \n" );
   printf ( "         Environmental and Water Resources Engineering.  \n" );
   printf ( "         Civil Engineering Department.                   \n" );
   printf ( "                                                         \n" );
# ifndef OS_LINUX
   printf ( "Windows version. Gnuplot extension disabled.             \n" );
# endif
# ifdef OS_LINUX
   printf ( "Linux version. Gnuplot extension enabled.                \n" );
   printf ( " . All graphs are made with Gnuplot 3.7.3.               \n" );
   printf ( " . The interface between C and Gnuplot is provided by    \n" );
   printf ( "   the gnuplot_i library written by  N. Devillard.       \n" );
# endif
   printf ( "------------------------------------------------------------\n" );
   printf ( "\n" );

} /* display_signature */


int handle_arguments ( int argc, char **argv )
/*
   # Arguments : Same as 'Main'
   #
   # Action : Verification of the number of arguments and
   #          check if the user asked for help.
   #
   # Return : 0 if an error occurred or the user asked for help. 1 otherwise.
*/
{
   if ( ( (argc == 2) && strncmp( argv[1], "--help", 6 ) ) || (argc < 2) )
   {
               printf ( "\n" );
      printf ( "Error.\nThe source file and the prefix must be specified.\n" );
      printf ( "\nUse the option \'--help\' for help" );
      printf ( "\n\n" );
               return 0;
   }

   /* Display help message if enquired by user*/
   if ( !strncmp( argv[1], "--help", 6 ) )
```

```
   {
      (void) display_help ();
      return 0;
   }

   if ( argc < 3 )
        {
                printf ( "\n" );
      printf ( "Error.\nThe source file and the prefix must be specified.\n" );
      printf ( "\nUse the option \'--help\' for help" );
      printf ( "\n\n" );
                return 0;
        }

   return 1;

} /* handle_arguments */


void display_main_menu ()
/*
   # Arguments : /
   #
   # Action : Display help.
   #
   # Return : /
*/
{
   printf ( "\nMAIN MENU\n" );
   printf ( "---------\n\n" );

   printf ( "-----------------------------------------------\n"   );
   printf ( "|   Process raw data ..................... 1  |\n"   );
   printf ( "|   Identify LWD ......................... 2  |\n"   );
   printf ( "|   Export processed data ................ 3  |\n"   );
# ifdef OS_LINUX
   printf ( "|   Plotting ............................. 4  |\n"   );
# endif
   printf ( "|                                             |\n"   );
   printf ( "|   Exit ................................. 9  |\n"   );
   printf ( "-----------------------------------------------\n"   );
   printf ( "\n" );

} /* display_main_menu */
```

```
/* ----------------------------------
 *
 *      raw.h
 *
 *      Laurent White
 *
 *      Date of creation : 2002-12-19
 *      Last update      : 2002-06-20
 *
 * ---------------------------------- */

# ifndef RAW_H
# define RAW_H

# include "defs.h"

int process_raw ( char *fname_raw, gps_position *gps_positions );
int read_raw ( char *fname_raw, gps_position *gps_positions );

int read_single_sounding_raw1 ( FILE *fp_raw, double *longitude, \
      double *latitude, double *depth, double *timestamp );
int read_single_sounding_raw2 ( FILE *fp_raw, double *longitude, \
      double *latitude, double *depth, double *timestamp );
int read_single_sounding_raw3 ( FILE *fp_raw, double *longitude, \
      double *latitude, double *depth, double *timestamp );

int compute_statistics ( gps_position *gps_positions, int n_gps_positions );
double compute_depth_mean ( gps_position *gps );
double compute_depth_variance ( gps_position *gps );
double compute_slope ( gps_position *gps1, gps_position *gps2 );
double compute_slope_mean ( gps_position *gps1, gps_position *gps2 );
double compute_slope_variance ( gps_position *gps1, gps_position *gps2 );
double compute_distance ( gps_position *gps1, gps_position *gps2 );

# endif
```

```
/* ------------------------------------------------------
 *
 *      raw.c
 *
 *      Laurent White
 *
 *      Date of creation : 2002-12-19
 *      Last update      : 2003-06-20
 *
 * ------------------------------------------------------ */

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

# include "defs.h"
# include "raw.h"

int process_raw ( char *fname_raw, gps_position *gps_positions )
/*
   # Arguments : fname_raw is the name of the raw data file.
   #           gps_positions is an array of (struct gps_position).
   #
   # Action : 1. Read all soundings, eliminate invalid ones and lump them into
   #             GPS position (using the structure gps_position)
   #          2. Compute all statistics for each GPS position. That is, mean depth
   #             depth variance, slope, mean slope (same as previous one) and slope
   #             variance.
   #
   # Return : Number of distinct GPS positions
   #          -1 if the user wants to return to the main menu.
   #          0 if an error occurred.
*/
{
   int n_gps_positions; /* Number of distinct gps positions */

   /* Read the file and lump it into distinct GPS positions */
   n_gps_positions = read_raw ( fname_raw, gps_positions );

   if ( n_gps_positions == -1 )
      return -1;

   if ( n_gps_positions == 0 )
      return 0;

   /* Compute all statistics for each GPS position */
   printf ( "   Computing statistics for each GPS position...\n" );
   (void) compute_statistics ( gps_positions, n_gps_positions );


   printf ( "Processing done. %d distinct GPS positions \
         have been detected.\n\n", n_gps_positions );

   /* Return the number of GPS positions */
   return n_gps_positions;

} /* process_raw */


int read_raw ( char *fname_raw, gps_position *gps_positions )
/*
   # Arguments : fp_raw is pointer to a stream.
   #           gps_positions is an array of (stuct gps_positions)
   #
   # Action : Reads all lines in the file pointed by fp_raw (according
   #          to the conversion string specified in 'read_single_sounding_rawN'),
   #          where N is the raw data ID number.
   #          Do not take into account records that are invalid. For each
   #          GPS position in the array, we update the longitude, latitude, number
   #          of soundings and array of depth soundings. Each GPS position is
   #          ready for statistics computation.
   #
   # Returns : Number of distinct GPS positions.
   #           Returns 0 if an error occurred.
   #           Returns -1 if the user wants to return to the Main Menu.
*/
{
   /* Variables */
   double   longitude;
   double   latitude;
   double   ref_longitude;    /* Reference longitude */
   double   ref_latitude;     /* Reference latitude */
   double   depth;
   double  timestamp;      /* Time stamp of sounding */
   int   valid;
   int   n_invalid_soundings; /* Number of invalid soundings */
   int   n_soundings;      /* Total number of soundings */
   int   n_distinct_positions;   /* Number of distinct GPS positions */
   int   n_local;    /* Counter of soundings for a given GPS location */
   int   index;
   int   choice;
   FILE    *fp_raw;     /* File pointer to raw data file */
```

```
/* Initializations */
n_invalid_soundings  = 0;
n_soundings          = 0;
n_distinct_positions = 0;
ref_longitude        = 1000.0;
ref_latitude         = 1000.0;  /* Set the reference position outside any
             * physical range so that the first sounding
             * is always a new position
             */

/* Open file */
fp_raw  = fopen ( fname_raw , "r" );
   if (fp_raw == NULL)
   {
      printf ( "Opening file '%s' failed.\n", fname_raw );
      return 0;
   }

/* Display menu */
printf ( "\nRAW DATA MENU\n" );
printf ( "-------------\n\n" );

printf ( "-------------------------------------------------------\n" );
printf ( "|   Sulphur River (May 2001) ..................... 1   |\n" );
printf ( "|   Sulphur River (January 2002) ................. 2   |\n" );
printf ( "|   Guadalupe River (April 2003) ................. 3   |\n" );
printf ( "|                                                       |\n" );
printf ( "|   Return to Main Menu .......................... 9   |\n" );
printf ( "-------------------------------------------------------\n" );
printf ( "\n" );

printf ( "    Selection : " );
scanf ( "%d", &choice );

if ( choice == 9 )
   return -1;

printf ( "\n" );
printf ( "\nProcessing all soundings in '%s'...\n", fname_raw );
printf ( "\n" );
printf ( "   Lumping soundings into common GPS positions..." );
fflush ( stdout );

/* Loop through all soundings */
while ( !feof(fp_raw) )
{
   /* Increment counter of all soundings */
   n_soundings++;

   switch ( choice )
   {
      case 1:
         valid = read_single_sounding_raw1 ( fp_raw, \
               &longitude, &latitude, &depth, &timestamp );
         if ( valid == -1 )
            return -1;
         break;

      case 2:
         valid = read_single_sounding_raw2 ( fp_raw, \
               &longitude, &latitude, &depth, &timestamp );
         if ( valid == -1 )
            return -1;
         break;

      case 3:
         valid = read_single_sounding_raw3 ( fp_raw, \
               &longitude, &latitude, &depth, &timestamp );
         if ( valid == -1 )
            return -1;
         break;

      case 0:
         return -1;
         break;

      default:
         return -1;
         break;
   }

   if ( valid )
   {
      if ( ( latitude == ref_latitude ) && ( longitude == ref_longitude ) )
      /* Same GPS position */
      {
         n_local++;
         gps_positions[index].n_soundings++;
         gps_positions[index].soundings[n_local-1] = depth;
         gps_positions[index].timestamps[n_local-1] = timestamp;

      }
      else
      /* New GPS position */
```

```
         {
            /* Increment the counter of distinct GPS positions */
            n_distinct_positions++;

            /* Current coordinates become reference coordinates */
            ref_longitude = longitude;
            ref_latitude  = latitude;

            /* Sets the number of soundings for that GPS position to 1 */
            n_local = 1;

            /* Set values for new GPS position */
            index = n_distinct_positions - 1;

            if (index > (MAX_GPS_POSITIONS - 1))
            {
               fprintf ( stderr, "Error. The index in \
                     'read_raw' is greater \
                     than the maximum allowed value.\n" );
               return -1;
            }
            gps_positions[index].id                    = n_distinct_positions;
            gps_positions[index].n_soundings           = 1;
            gps_positions[index].longitude             = ref_longitude;
            gps_positions[index].latitude              = ref_latitude;
            gps_positions[index].soundings[n_local-1]  = depth;
            gps_positions[index].timestamps[n_local-1] = timestamp;

            /* Initialization of all statistics */
            gps_positions[index].depth_mean = 0.0;
            gps_positions[index].depth_var  = 0.0;
            gps_positions[index].slope      = 0.0;
            gps_positions[index].slope_mean = 0.0;
            gps_positions[index].slope_var  = 0.0;
            gps_positions[index].distance   = 0.0;
         }
      }
      else
         /* Increment counter of invalid soundings */
         n_invalid_soundings++;

   } /* end fp_raw */

   printf ( " (Detected %d invalid soundings out of %d soundings).\n", \
         n_invalid_soundings, n_soundings );

   /* Close stream */
   fclose ( fp_raw );

   return n_distinct_positions;

} /* read_raw */


int read_single_sounding_raw1 ( FILE *fp_raw, double *longitude, \
      double *latitude, double *depth, double *timestamp )
/*
   # Arguments : fp_raw : pointer to a stream.
   #      longitude, latitude, depth : data associated with current sounding.
   #             The file MUST contain data using the format of SULPHUR RIVER SITE 1.
   #
   # Action : Reads a line in the file pointed by fp_raw and
   #          assigns values to longitude,
   #          latitude and depth.
   #
   # Return : A non-zero integer if the sounding is valid.
   #          In this case, the arguments 'longitude', 'latitude'
   #          and 'depth' have well-defined values.
   #
   #          0 if the sounding is invalid.
   #         -1 if a reading error occurred.
   #          In this case, the arguments 'longitude',
   #          'latitude' and 'depth' have NULL values.

*/
{
   /* Variable starting with underscore are used to scan the file */
   int         _day;
   int         _month;
   int         _year;
   int         _hour;
   int         _min;
   double      _sec;
   double      _depth;
   int         _quality;
   double      _tranducer_depth;
   int         _speed_sound;
   double      _lat_deg;
   double      _lat_min;
   double      _long_deg;
   double      _long_min;
   char        _lat;
   char        _long;
   int         _tmp;
```

```
   int         n_read;

   int       valid;

   /* Read the sounding (the nasty conversion string will convert raw1 data ONLY !!) */
        n_read = fscanf ( fp_raw, \
         "%2d%2d%4d,%2d%2d%6lf,HF,%8lf,%d,  %4lf,%4d,%5d ,%2lf %9lf%1c,%3lf %9lf%1c\n",
                         &_day, &_month, &_year, &_hour, &_min, &_sec, &_depth, &_quality,
                         &_tranducer_depth, &_speed_sound, &_tmp,
                         &_lat_deg, &_lat_min, &_lat, &_long_deg, &_long_min, &_long );

   if ( n_read != 17 )
   {
      printf ( "\nError while reading raw data file !\n" );
      return -1;
   }

   /*
    * Check for validity of sounding : a quality of 1 indicates
    * a valid sounding. Also, the latitude must be North and
    * the longitude must be west.
    */
   valid = ( _quality == 1 ) && ( _lat == LAT_C_1 ) && ( _long == LONG_C_1 );

   /*
    * If the degree part of the coordinate is 0, set it to
    * the reference location.
    */
   if ( _lat_deg == 0 )
      _lat_deg = LATITUDE_1;

   if ( _long_deg == 0 )
      _long_deg = LONGITUDE_1;

   /* Set the values of the arguments */
   *longitude = _long_deg + _long_min / 60.0;
   *latitude  = _lat_deg  + _lat_min  / 60.0;
   *depth     = _depth / CONV_FEET_M;
   *timestamp = (double) (_hour*3600 + _min*60 + _sec);

   /*
    * Change longitude to its opposite if it is 'West' longitude.
    * Idem for latitude if it's 'South' latitude
   */
   if ( _long == 'W' )
      *longitude *= -1;

   if ( _lat == 'S' )
      *latitude *= -1;

   /* If sounding is invalid, set NULL to arguments */
   if ( !valid )
   {
      longitude = NULL;
      latitude  = NULL;
      depth     = NULL;
   }

   return valid;

} /* read_single_sounding_raw1 */


int read_single_sounding_raw2 ( FILE *fp_raw, double *longitude, \
      double *latitude, double *depth, double *timestamp )
/*
   # Arguments : fp_raw : pointer to a stream.
   #      longitude, latitude, depth : data associated with current sounding.
   #             The file MUST contain data using the format of SULPHUR RIVER SITE 2.
   #
   # Action : Reads a line in the file pointed by fp_raw and assigns values to longitude,
   #          latitude and depth.
   #
   # Return : A non-zero integer if the sounding is valid.
   #          In this case, the arguments 'longitude',
   #          'latitude' and 'depth' have well-defined values.
   #
   #          0 if the sounding is invalid.
   #          -1 if a reading error occurred.
   #          In this case, the arguments 'longitude',
   #          'latitude' and 'depth' have NULL values.

*/
{
   /* Variable starting with underscore are used to scan the file */
   int         _day;
   int         _month;
   int         _year;
   int         _hour;
   int         _min;
   double      _sec;
   double      _depth;
   int         _quality;
   double      _lat_deg;
```

```
   double      _lat_min;
   double      _long_deg;
   double      _long_min;
   char        _lat;
   char        _long;
   int         _latency;
   int         n_read;

   int       valid;

   /* Read the sounding (the nasty conversion string will convert raw2 data ONLY !!) */
   n_read = fscanf ( fp_raw, \
         "%2d%2d%4d,%2d%2d%6lf,%8lf,%d,%2lf %9lf%1c,%3lf %9lf%1c,%4d\n",
                        &_day, &_month, &_year, &_hour, &_min, &_sec, &_depth, &_quality,
                        &_lat_deg, &_lat_min, &_lat, &_long_deg, &_long_min, &_long,
          &_latency );

   if ( n_read != 15 )
   {
      printf ( "\nError while reading raw data file !\n" );
      return -1;
   }

   /*
    * Check for validity of sounding : a quality of 1 indicates
    * a valid sounding. Also, the latitude must be North and
    * the longitude must be west.
    */
   valid = ( _quality == 1 ) && ( _lat == LAT_C_2 ) && ( _long == LONG_C_2 );

   /*
    * If the degree part of the coordinate is 0, set it to
    * the reference location.
    */
   if ( _lat_deg == 0 )
      _lat_deg = LATITUDE_2;

   if ( _long_deg == 0 )
      _long_deg = LONGITUDE_2;

   /* Set the values of the arguments */
   *longitude = _long_deg + _long_min / 60.0;
   *latitude  = _lat_deg  + _lat_min  / 60.0;
   *depth     = _depth;
   *timestamp = (double) (_hour*3600 + _min*60 + _sec);

   /*
    * Change longitude to its opposite if it is 'West' longitude.
    * Idem for latitude if it's 'South' latitude
   */
   if ( _long == 'W' )
      *longitude *= -1;

   if ( _lat == 'S' )
      *latitude *= -1;

   /* If sounding is invalid, set NULL to arguments */
   if ( !valid )
   {
      longitude = NULL;
      latitude  = NULL;
      depth     = NULL;
   }

   return valid;

} /* read_single_sounding_raw2 */


int read_single_sounding_raw3 ( FILE *fp_raw, double *longitude, \
      double *latitude, double *depth, double *timestamp )
/*
   # Arguments : fp_raw : pointer to a stream.
   #      longitude, latitude, depth : data associated with current sounding.
   #             The file MUST contain data using the format of GUADALUPE RIVER.
   #
   # Action : Reads a line in the file pointed by fp_raw and assigns values to longitude,
   #          latitude and depth.
   #
   # Return : A non-zero integer if the sounding is valid.
   #          In this case, the arguments 'longitude',
   #          'latitude' and 'depth' have well-defined values.
   #
   #          0 if the sounding is invalid.
   #          -1 if a reading error occurred.
   #          In this case, the arguments 'longitude',
   #          'latitude' and 'depth' have NULL values.

*/
{
   /* Variable starting with underscore are used to scan the file */
   int         _day;
   int         _month;
   int         _year;
```

```
   int         _hour;
   int         _min;
   double      _sec;
   double      _depth;
   int         _quality;
   double      _tranducer_depth;
   double      _lat_deg;
   double      _lat_min;
   double      _long_deg;
   double      _long_min;
   char        _lat;
   char        _long;
   int         _tmp;
   int         _latency;
   int         n_read;

   int       valid;

   /* Read the sounding (the nasty conversion string will convert raw3 data ONLY !!) */
   n_read = fscanf ( fp_raw, \
         "%2d%2d%4d,%2d%2d%2d%6lf,%5d,%8lf,%d,  %4lf,%2lf %9lf%1c,%3lf %9lf%1c,%4d\n",
                        &_day, &_month, &_year, &_hour, &_min,
          &_sec, &_tmp, &_depth, &_quality,
                        &_tranducer_depth,
                        &_lat_deg, &_lat_min, &_lat,
          &_long_deg, &_long_min, &_long,
          &_latency );

   if ( n_read != 17 )
   {
      printf ( "\nError while reading raw data file !\n" );
      return -1;
   }

   /*
    * Check for validity of sounding : a quality of 1 indicates
    * a valid sounding. Also, the latitude must be North and
    * the longitude must be west.
    */
   valid = ( _quality == 1 ) && ( _lat == LAT_C_3 ) && ( _long == LONG_C_3 );

   /*
    * If the degree part of the coordinate is 0, set it to
    * the reference location.
    */
   if ( _lat_deg == 0 )
      _lat_deg = LATITUDE_3;

   if ( _long_deg == 0 )
      _long_deg = LONGITUDE_3;

   /* Set the values of the arguments */
   *longitude = _long_deg + _long_min / 60.0;
   *latitude  = _lat_deg  + _lat_min  / 60.0;
   *depth     = _depth;
   *timestamp = (double) (_hour*3600 + _min*60 + _sec);

   /*
    * Change longitude to its opposite if it is 'West' longitude.
    * Idem for latitude if it's 'South' latitude
   */
   if ( _long == 'W' )
      *longitude *= -1;

   if ( _lat == 'S' )
      *latitude *= -1;

   /* If sounding is invalid, set NULL to arguments */
   if ( !valid )
   {
      longitude = NULL;
      latitude  = NULL;
      depth     = NULL;
   }

   return valid;

} /* read_single_sounding_raw3 */


int compute_statistics ( gps_position *gps_positions, int n_gps_positions )
/*
   #
   #
   # Return : 1 no error was encountered while calculating the statistics.
   #          0 if an error occurred.
*/
{
   int    i;
   double dist;

   /* Compute mean depth and depth variance */
   for ( i = 0 ; i < n_gps_positions ; i++ )
   {
```

```
      (void) compute_depth_mean      ( gps_positions + i );
      (void) compute_depth_variance  ( gps_positions + i );
   }

   /* Compute mean slope and slope variance */
   for ( i = 0 ; i < (n_gps_positions - 1) ; i++ )
   {
      dist = compute_distance ( gps_positions + i, gps_positions + i + 1 );

      if ( dist > 0.0 )
      {
         (void) compute_slope ( gps_positions + i, gps_positions + i + 1 );
         (void) compute_slope_mean ( gps_positions + i, gps_positions + i + 1 );
         (void) compute_slope_variance ( gps_positions + i, gps_positions + i + 1 );
      }
      else
         fprintf ( stderr, \
               "      Distance (%f) between GPS%d and GPS%d is \
               negative or equal to 0. GPS%d will be skiped.\n",\
               dist, i, i+1, i );

   }

   return 1;

} /* compute_statistics */


double compute_depth_mean ( gps_position *gps )
/*
   # Arguments : gps is a pointer to (struct gps_position),
   #             representative of a valid GPS position.
   #
   # Action : Compute the mean of all depths at GPS position. Assign it to gps.
   #
   # Return : mean of all depths at GPS position.
*/
{
   int N;          /* Number of soundings */
   int i;          /* Counter */
   double sum_depth; /* Running sum of all depths */

   /* Retrieve the number of soundings for the GPS position */
   N = (*gps).n_soundings;

   /* Initialiazes the sum of depths */
   sum_depth = 0.0;

   for ( i = 0 ; i < N ; i++ )
      sum_depth += (*gps).soundings[i];

   /* Compute the mean depth, assign it to GPS position
    * and set it as return value (vive le C !) */
   return ( (*gps).depth_mean = (sum_depth / N) );

} /* compute_depth_mean */


double compute_depth_variance ( gps_position *gps )
/*
   # Arguments : gps is a pointer to (struct gps_position),
   #             representative of a valid GPS position.
   #
   # Action : Compute the variance of all depths at GPS position and assign it to gps.
   #
   # Return : variance of all depths at GPS position.
*/
{
   int N;
   int i;
   double depth_mean;      /* Mean depth of GPS position */
   double sum_squared_difference;   /* Running sum of (h_i - h_avg)^2 */

   /* Retrieve the number of soundings and the mean depth for the GPS position */
   N          = (*gps).n_soundings;
   depth_mean = (*gps).depth_mean;

   /* Initialiazes the running sum */
   sum_squared_difference = 0.0;

   for ( i = 0 ; i < N ; i++ )
      sum_squared_difference += pow( (*gps).soundings[i] - depth_mean, 2 );

   /* Compute the variance, assign it to GPS position
    * and set it as return value (vive le C !) */
   return ( (*gps).depth_var = (sum_squared_difference / N) );

} /* compute_depth_variance */


double compute_slope ( gps_position *gps1, gps_position *gps2 )
/*
   # Arguments : gps1 and gps2 are pointers to (struct gps_position),
   #             representative of valid GPS positions.
```

```
   #
   # Action : Calculate the slope between GPS positions using the mean depths
   #          at each position in the calculation. Assign it to gps1.
   #
   # Return : Slope between GPS positions using the mean depths in the calculation.
*/
{
   double dist;
   double mean_depth1;
   double mean_depth2;

   dist        = (*gps1).distance;     /* Distance between GPS1 and GPS2 */

   if (dist == 0.0)
   {
      fprintf ( stderr, \
            "Error. Distance between GPS%d and GPS%d \
            must be computed before computing the slope.\n",\
             (*gps1).id, (*gps2).id );
      return 0;
   }

   mean_depth1 = (*gps1).depth_mean;
   mean_depth2 = (*gps2).depth_mean;

   return ( (*gps1).slope = (mean_depth2 - mean_depth1) / dist );

} /* compute_slope */


double compute_slope_mean ( gps_position *gps1, gps_position *gps2 )
/*
   # Arguments : gps1 and gps2 are pointers to (struct gps_position),
   #             representative of valid GPS positions.
   #
   # Action : Calculate the mean of all possible slopes between both GPS positions
   #          and assign it to gps1.
   #
   # Return : Mean of all possible slopes between both GPS positions.
*/
{
   int N1;
   int N2;
   double dist;      /* Distance between GPS positions */
   int i;
   int j;
   double sum_slope; /* Running sum of all possible slopes */

   /* Distance between GPS positions */
   dist = (*gps1).distance;

   if (dist == 0.0)
   {
      fprintf ( stderr, \
            "Error. Distance between GPS%d and GPS%d \
            must be computed before computing the slope.\n",
                        (*gps1).id, (*gps2).id );
      return 0;
   }


   /* Retrieve the number of soundings at each GPS position */
   N1 = (*gps1).n_soundings;
   N2 = (*gps2).n_soundings;

   /* Compute sum of all possible slopes */
   sum_slope = 0.0;
   for ( i = 0 ; i < N1 ; i++ )
      for ( j = 0 ; j < N2 ; j++ )
         sum_slope += ( (*gps2).soundings[j] - (*gps1).soundings[i] ) / dist;

   return ( (*gps1).slope_mean = sum_slope / (N1*N2) );

} /* compute_slope_mean */


double compute_slope_variance ( gps_position *gps1, gps_position *gps2 )
/*
   # Arguments : gps1 and gps2 are pointers to (struct gps_position),
   #             representative of valid GPS positions.
   #
   # Action : Calculate the variance of all possible slopes between both GPS positions
   #          and assign it to gps1.
   #
   # Return : Variance of all possible slopes between both GPS positions.
*/
{
   int N1;
   int N2;
   double dist;         /* Distance between GPS positions */
   int i;
   int j;
   double slope_ij;     /* Slope between sounding i
                  of GPS1 and sounding j of GPS2 */
```

```
   double slope_mean;      /* Mean of all slopes between GPS1 and GPS2 */
   double sum_squared_difference;   /* Running sum of all possible slopes */

   /* Distance between GPS positions */
   dist = (*gps1).distance;

   if (dist == 0.0)
   {
      fprintf ( stderr, \
            "Error. Distance between GPS%d and GPS%d \
            must be computed before computing the slope.\n",
                        (*gps1).id, (*gps2).id );
      return 0;
   }


   /* Mean of all slopes between GPS positions */
   slope_mean = (*gps1).slope_mean;

   /* Retrieve the number of soundings at each GPS position */
   N1 = (*gps1).n_soundings;
   N2 = (*gps2).n_soundings;

   /* Compute sum of all possible slopes */
   sum_squared_difference = 0.0;
   for ( i = 0 ; i < N1 ; i++ )
      for ( j = 0 ; j < N2 ; j++ )
      {
         slope_ij = ( (*gps2).soundings[j] - (*gps1).soundings[i] ) / dist;
         sum_squared_difference += pow ( slope_ij - slope_mean, 2 );
      }

   return ( (*gps1).slope_var = sum_squared_difference / (N1*N2) );


} /* compute_slope_variance */


double compute_distance ( gps_position *gps1, gps_position *gps2 )
/*
   # Arguments : gps1 and gps2 are pointers to (struct gps_position),
   #             representative of valid GPS positions between which
   #             the distance is to be calculated.
   #
   # Action : Calculate the distance between both GPS positions and assign it
   #          to gps1.
   #
   # Return : Distance between GPS positions.
*/
{

   double lambda1;      /* GPS1 - longitude (rad.) */
   double phi1;      /* GPS1 - latitude  (rad.) */
   double lambda2;      /* GPS2 - longitude (rad.) */
   double phi2;      /* GPS2 - latitude  (rad.) */

   double x1,y1,z1;  /* GPS1 - Cartesian coordinates */
   double x2,y2,z2;  /* GPS2 - Cartesian coordinates */

   double L_sq;      /* Distance of straight line
               between (x1,y1,z1) and (x2,y2,z2) */
   double alpha;     /* Angle between radii defined by GPS1 and GPS2 */


   /* Retrieve spherical coordinates (M_PI is defined in math.h) */
   lambda1 = (*gps1).longitude * M_PI / 180.0;
   phi1    = (*gps1).latitude  * M_PI / 180.0;

   lambda2 = (*gps2).longitude * M_PI / 180.0;
   phi2    = (*gps2).latitude  * M_PI / 180.0;


   /* Conversion to Cartesian coordinates (RE = Earth radius) */
   x1 = RE * cos ( phi1 ) * cos ( lambda1 );
   y1 = RE * cos ( phi1 ) * sin ( lambda1 );
   z1 = RE * sin ( phi1 );

   x2 = RE * cos ( phi2 ) * cos ( lambda2 );
   y2 = RE * cos ( phi2 ) * sin ( lambda2 );
   z2 = RE * sin ( phi2 );


   /* Distance (squared) between positions */
   L_sq = pow ( x1-x2, 2 ) + pow ( y1-y2, 2 ) + pow ( z1-z2, 2 );


   /* Compute the angle between both positions (Generalized Pythagoras) */
   alpha = acos( 1.0 - L_sq / (2.0*pow(RE,2)) );

   return ( (*gps1).distance = alpha*RE );

} /* compute_distance */
```

```
/* ----------------------------------
 *
 *      postprocess.h
 *
 *      Laurent White
 *
 *      Date of creation : 2002-12-19
 *
 * ---------------------------------- */

# ifndef POSTPROCESS_H
# define POSTPROCESS_H

# include "defs.h"

int export_to_dx ( char *basename, int items );

int export_to_matlab ( char *prefix, gps_position *gps_positions, \
      int n_gps_positions );

int export_to_sms ( char *prefix, gps_position *gps_positions, \
      int n_gps_positions );

# endif /* POSTPROCESS_H */
```

```
/* ----------------------------------
 *
 *      postprocess.c
 *
 *      Laurent White
 *
 *      Date of creation : 2002-12-20
 *
 * ---------------------------------- */

# include <stdio.h>
# include <stdlib.h>
# include <string.h>

# include "postprocess.h"

int export_to_dx ( char *basename, int items )
/*
   # input_file : file basename containing the data to be exported into a DX format
   # items      : number of items in the file (number of positions)
*/
{

        FILE *fp_in;
        FILE *fp_pos;
        FILE *fp_dat;
        FILE *fp_dx;
        char fname_in[100];     /* File with processed data */
        char fname_pos[100];    /* DX file : positions */
        char fname_dat[100];    /* DX file : data */
        char fname_dx[100];     /* DX file : field structure */

        int _day;
        int _month;
        int _year;
        double _time;
        double _latitude;
        double _longitude;
        double _depth;
   double _var;
        int _tmp;

        /* ------------------------
         * --- Opens the files ---
         * ------------------------ */
        strcpy ( fname_in, basename );
        strcat ( fname_in, ".pro" );
        fp_in = fopen ( fname_in, "r" );
        if (fp_in == NULL)
        {
                printf ( "Opening file `%s' failed.\n", fname_in );
                return 0;
        }

        strcpy ( fname_pos, basename );
        strcat ( fname_pos, ".pos" );
        fp_pos = fopen ( fname_pos, "w" );
        if (fp_pos == NULL)
        {
                printf ( "Opening file `%s' failed.\n", fname_pos );
                return 0;
        }

        strcpy ( fname_dat, basename );
        strcat ( fname_dat, ".dat" );
        fp_dat = fopen ( fname_dat, "w" );
        if (fp_dat == NULL)
        {
                printf ( "Opening file `%s' failed.\n", fname_dat );
                return 0;
        }

        strcpy ( fname_dx, basename );
        strcat ( fname_dx, ".dx" );
        fp_dx = fopen ( fname_dx, "w" );
        if (fp_dx == NULL)
        {
                printf ( "Opening file `%s' failed.\n", fname_dx );
                return 0;
        }

        /* ------------------------------
         * --- Writes to the dx files ---
         * ------------------------------ */


        while ( !feof(fp_in) )
        {
                fscanf ( fp_in, "%2d-%2d-%4d %lf %lf %lf %lf %lf %d\n",
               &_day, &_month, &_year, &_time,
          &_latitude, &_longitude, &_depth, &_var, &_tmp );

                fprintf ( fp_pos, "%.10f %.10f\n", _longitude, _latitude );
                fprintf ( fp_dat, "%lf\n", _depth );
```

```
        }

        fprintf ( fp_dx, "# POSITIONS\n" );
        fprintf ( fp_dx, "object 1 class array type float rank 1 shape 2 items %d\n",
         items );
        fprintf ( fp_dx, "data file %s,0\n\n", fname_pos );

        fprintf ( fp_dx, "# DATA\n" );
        fprintf ( fp_dx, "object 2 class array type float rank 0 items %d\n", items );
        fprintf ( fp_dx, "data file %s,0\n", fname_dat );
        fprintf ( fp_dx, "attribute \"dep\" string \"positions\"\n\n" );

        fprintf ( fp_dx, "# FIELD\n" );
        fprintf ( fp_dx, "object \"irregular positions\" class field\n" );
        fprintf ( fp_dx, "component \"positions\" value 1\n" );
        fprintf ( fp_dx, "component \"data\" value 2\n" );


        /* Closes files */
        fclose ( fp_in );
        fclose ( fp_pos );
        fclose ( fp_dat );
        fclose ( fp_dx );
        return -1;

} /* export_to_dx */


int export_to_matlab ( char *prefix, gps_position *gps_positions, int n_gps_positions )
/*
   # prefix          : prefix of filenames used to export data to use under Matlab.
   # gps_positions   : array of GPS positions (struct gps_position), each one containing
   #                   all data associated with it
   #                   (mean depth, distance to next position, ...)
   # n_gps_positions : number of distinct GPS positions (number of elements in the array)
   #
   # ACTION : Export the data into three files :
   #          'prefix.sum', 'prefix.lum' and 'prefix.fin'
   #          'prefix.sum' contains summarized data for each GPS position.
   #          'prefix.lum' is made up of a series of arrays numbered GPS1 to GPSN (where
   #          N = n_gps_positions). Each array contains all depth measurements associated
   #          with the GPS position.
   #          'prefix.fin' is a fine version of the bathymetry obtained by using ALL
   #          depth measurements (spread equidistantly between adjacent GPS positions).
*/
{
   /* Variables */
   FILE *fp_lum;
   FILE *fp_sum;
   FILE *fp_ori;
   FILE *fp_fin;

   char fname_lum[128]; /* Filename of file containing lumped data */
   char fname_sum[128];    /* Filename of file containing summarized data */
   char fname_ori[128]; /* Filename of file containing fine bathymetry -- not for MATLAB */
   char fname_fin[128]; /* Filename of file containing fine bathymetry */

   int i;
   int j;
   int n_skipped = 0;

   double ksi;    /* Curvilinear coordinate */
   int    n_local;      /* Number of soundings at current GPS position */
   double delta_ksi; /* Local increment in curvilinear coordinate */
   double delta_x;      /* Local increment in longitude */
   double delta_y;      /* Local increment in latitude */
   double x;
   double y;


   if ( n_gps_positions <= 0 )
   {
      printf ( "\nNo data to export...\n" );
      printf ( "Either an error occurred while processing the raw data\n" );
      printf ( "or no data have been found. Make sure to process the data first !\n" );
      printf ( "Aborting ...\n" );
      return 0;
   }

   /* ------------------------------------------------------------------- */
   /* I/O management */
   /* ------------------------------------------------------------------- */

   strcpy ( fname_lum, prefix );
   strcat ( fname_lum, "_lum.m" );
        fp_lum = fopen ( fname_lum, "w" );
        if (fp_lum == NULL)
        {
                printf ( "Opening file '%s' failed in 'export_to_matlab'.\n", fname_lum );
                return 0;
        }

   strcpy ( fname_sum, prefix );
   strcat ( fname_sum, "_sum.m" );
```

```
     fp_sum = fopen ( fname_sum, "w" );
     if (fp_sum == NULL)
     {
             printf ( "Opening file '%s' failed in 'export_to_matlab'.\n", fname_sum );
             return 0;
     }

strcpy ( fname_ori, prefix );
strcat ( fname_ori, ".ori" );
     fp_ori = fopen ( fname_ori, "w" );
     if (fp_ori == NULL)
     {
             printf ( "Opening file '%s' failed in 'export_to_matlab'.\n", fname_ori );
             return 0;
     }

strcpy ( fname_fin, prefix );
strcat ( fname_fin, "_fin.m" );
     fp_fin = fopen ( fname_fin, "w" );
     if (fp_fin == NULL)
     {
             printf ( "Opening file '%s' failed in 'export_to_matlab'.\n", fname_fin );
             return 0;
     }

printf ( "\nExporting processed data for use under Matlab...\n" );


/* ------------------------------------------------------------------- */
/* Exportation of summarized data (prefix.sum) */
/* ------------------------------------------------------------------- */

printf ( "   Exporting summarized data to '%s'\n", fname_sum );

fprintf ( fp_sum, "%% X,Y     : Cartesian coordinates.\n" );
fprintf ( fp_sum, "%% HM,HV   : Mean depth and variance of depth.\n" );
fprintf ( fp_sum, "%% S,SM,SV : Slope, mean slope (same as previous), \
      variance of slope.\n" );
fprintf ( fp_sum, "%% T       : Time stamps.\n" );
fprintf ( fp_sum, "%% N       : Number of soundings at each position.\n" );

     fprintf ( fp_sum, "DATA = [\n" );

for ( i = 0 ; i < n_gps_positions ; i++ )
{
        if ( (gps_positions[i].distance > 0) || (i == (n_gps_positions-1)) )
      fprintf ( fp_sum, "%.10f %.10f %f %f %f %f %f %f %d\n",
             gps_positions[i].longitude, gps_positions[i].latitude,
        gps_positions[i].depth_mean, gps_positions[i].depth_var,
        gps_positions[i].slope,
        gps_positions[i].slope_mean,
        gps_positions[i].slope_var,
        gps_positions[i].timestamps[0], gps_positions[i].n_soundings );
   else
      n_skipped++;

}

fprintf ( fp_sum, "];\n\n" );

fprintf ( fp_sum, "X  = DATA(:,1);\n" );
fprintf ( fp_sum, "Y  = DATA(:,2);\n" );
fprintf ( fp_sum, "HM = DATA(:,3);\n" );
fprintf ( fp_sum, "HV = DATA(:,4);\n" );
fprintf ( fp_sum, "S  = DATA(1:%d,5);\n", n_gps_positions - 1 - n_skipped );
fprintf ( fp_sum, "SM = DATA(1:%d,6);\n", n_gps_positions - 1 - n_skipped );
fprintf ( fp_sum, "SV = DATA(1:%d,7);\n", n_gps_positions - 1 - n_skipped );
fprintf ( fp_sum, "T  = DATA(:,8);\n" );
fprintf ( fp_sum, "N  = DATA(:,9);\n" );

fprintf ( fp_sum, "clear DATA;\n" );

printf ( "      %d GPS position(s) has(have) been skipped.\n", n_skipped );


/* ------------------------------------------------------------------- */
/* Exportation of lumped data (prefix.lum) */
/* ------------------------------------------------------------------- */

printf ( "   Exporting lumped data to '%s'\n\n", fname_lum );

for ( i = 0 ; i < n_gps_positions ; i++ )
{
   fprintf ( fp_lum, "%% (%.10f,%.10f)\n",
         gps_positions[i].longitude, gps_positions[i].latitude );

   fprintf ( fp_lum, "GPS%d = [\n", gps_positions[i].id );
   for ( j = 0 ; j < gps_positions[i].n_soundings ; j++ )
      fprintf ( fp_lum, "%.8f \n", gps_positions[i].soundings[j] );

   fprintf ( fp_lum, "];\n\n");
}
```

```
   /* ------------------------------------------------------------------- */
   /* Exportation of fine bathymetry (prefix.fin) */
   /* ------------------------------------------------------------------- */

   printf ( "   Exporting fine bathymetry to '%s'\n\n", fname_fin );

   fprintf ( fp_fin, "%% X       : Longitude [deg.].\n" );
   fprintf ( fp_fin, "%% Y       : Latitude [deg.].\n" );
   fprintf ( fp_fin, "%% KSI     : Curvilinear coordinates.\n" );
   fprintf ( fp_fin, "%% H       : Local depth measurement.\n" );
   fprintf ( fp_fin, "%% HM      : Mean depth measurement.\n" );
   fprintf ( fp_fin, "%% T       : Timestamps.\n" );

        fprintf ( fp_fin, "DATA = [\n" );

   ksi = 0.0;

   for ( i = 0 ; i < (n_gps_positions - 1) ; i++ )
   {
      /* Current GPS position */
      x = gps_positions[i].longitude;
      y = gps_positions[i].latitude;

      /* Retrieve the number of soudings for the current GPS position */
      n_local = gps_positions[i].n_soundings;

      /* Compute the increment in ksi. Note : we divide by n_local and not
         (n_local-1) because the last sounding associated with the current GPS
         position must not lie on the next one */
      delta_ksi = gps_positions[i].distance / n_local;
      delta_x   = (gps_positions[i+1].longitude - x) / n_local;
      delta_y   = (gps_positions[i+1].latitude  - y) / n_local;

      for ( j = 0 ; j < n_local ; j++ )
      {
         fprintf ( fp_fin, "%.10f %.10f %.8f %.8f %.8f %.3f\n",
                   x, y, ksi,
              gps_positions[i].soundings[j],
              gps_positions[i].depth_mean,
              gps_positions[i].timestamps[j] );

         fprintf ( fp_ori, "%.10f, %.10f, %.10f\n",
               x, y, gps_positions[i].soundings[j] );
         ksi += delta_ksi;
         x   += delta_x;
         y   += delta_y;
      }
   }
   fprintf ( fp_fin, "];\n\n" );

   fprintf ( fp_fin, "X    = DATA(:,1);\n" );
   fprintf ( fp_fin, "Y    = DATA(:,2);\n" );
   fprintf ( fp_fin, "KSI  = DATA(:,3);\n" );
   fprintf ( fp_fin, "H    = DATA(:,4);\n" );
   fprintf ( fp_fin, "HM   = DATA(:,5);\n" );
   fprintf ( fp_fin, "T    = DATA(:,6);\n" );

   fprintf ( fp_fin, "clear DATA;\n" );


   /* ------------------------------------------------------------------- */
   /* Close streams */
   /* ------------------------------------------------------------------- */

   fclose ( fp_lum );
   fclose ( fp_sum );
   fclose ( fp_ori );
   fclose ( fp_fin );

   return -1;

} /* export_to_matlab */


int export_to_sms ( char *prefix, gps_position *gps_positions, int n_gps_positions )
/*
   # prefix          : prefix of filenames used to export data to use under Matlab.
   # gps_positions   : array of GPS positions (struct gps_position), each one containing
   #                   all data associated with it
   #                   (mean depth, distance to next position, ...)
   # n_gps_positions : number of distinct GPS positions (number of elements in the array)
*/
{
   /* Variables */
   FILE *fp_sms;

   char fname_sms[128]; /* Filename of file containing lumped data */

   int i;
   int n_skipped = 0;

   /* ------------------------------------------------------------------- */
   /* I/O management */
   /* ------------------------------------------------------------------- */
```

```
   strcpy ( fname_sms, prefix );
   strcat ( fname_sms, "_sms.m" );
        fp_sms = fopen ( fname_sms, "w" );
        if (fp_sms == NULL)
        {
                printf ( "Opening file '%s' failed in 'export_to_SMS'.\n", fname_sms );
                return 0;
        }

   printf ( "\nExporting processed data for use under SMS...\n" );


   /* ------------------------------------------------------------------ */
   /* Exportation of summarized data (prefix.sum) */
   /* ------------------------------------------------------------------ */

   printf ( "   Exporting summarized data to '%s'\n", fname_sms );

   fprintf ( fp_sms, "%% X,Y      : Cartesian coordinates.\n" );
   fprintf ( fp_sms, "%% HM       : Mean depth.\n" );

   for ( i = 0 ; i < n_gps_positions ; i++ )
   {
          if ( (gps_positions[i].distance > 0) || (i == (n_gps_positions-1)) )
        fprintf ( fp_sms, "%.10f , %.10f , %f\n",
                gps_positions[i].longitude, gps_positions[i].latitude,
          gps_positions[i].depth_mean );
      else
         n_skipped++;

   }

   printf ( "      %d GPS position(s) has(have) been skipped.\n", n_skipped );

   fclose ( fp_sms );

   return -1;

} /* export_to_sms */
```

```
/* ------------------------------------------------------
 *
 *      medianfilter.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-16
 *      Last update      : 2003-06-17
 *
 * ------------------------------------------------------ */

# ifndef MEDIANFILTER_H
# define MEDIANFILTER_H

int median_filter ( double *H_ori, double *H_flt, int length, int N );

double median ( double *H, int length );

void bubble_sort ( double *H, int length );

# endif /* MEDIANFILTER_H */
```

```
/* ------------------------------------------------------
 *
 *      medianfilter.c
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-16
 *      Last update      : 2003-06-17
 *
 *      NOTE : in the arguments, 'length' ALWAYS refers to
 *             the number of elements in the array that
 *             is passed as argument. And N ALWAYS refers
 *             to half the length of the median filter ;
 *        the length being (2*N + 1).
 * ------------------------------------------------------ */

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

# include "defs.h"
# include "medianfilter.h"


int median_filter ( double *H_ori, double *H_flt, int length, int N )
/*
   # Arguments : H_ori is the original signal.
   #             H_flt is the filtered signal (returned by the function).
   #             length is the number of elemnts in H_ori and H_flt.
   #             N is half the length of the filter.
   #
   # Action : Filter the original signal H_ori. The filtered signal is
   #          H_flt.
   #
   # Return : 0 if an error occurred. 1 otherwise.
*/
{
   int    i;
   double *H_temp;

   /* Create appended array */
   H_temp = (double *) malloc ( (length + 2*N) * sizeof ( double ) );
   if ( H_temp == NULL )
   {
      printf ( "Memory allocation failure in 'median_filter'.\n" );
      return 0;
   }

   /*
    * Append H_temp with N * H_ori[0] and N*H_ori[length], i.e.
    * the first N values of H_temp are filled with H_ori[0] and
    * the N last values of H_temp are filled with H_ori[length].
    */
   for ( i = 0 ; i < N ; i++ )
   {
      H_temp[i]                    = H_ori[0];
      H_temp[length + 2*N - 1 - i] = H_ori[length-1];
   }

   /* Copy H_ori to the central part of H_temp */
   for ( i = N ; i < (length + N) ; i++ )
      H_temp[i] = H_ori[i-N];

   /* Median filtering */
   for ( i = 0 ; i < length ; i++ )
   {
      H_flt[i] = median ( H_temp + i , 2*N+1 );
   }


   return 1;

} /* median_filter */


double median ( double *H, int length )
/*
   # Arguments : 'H' is an array of 'length' elements.
   #
   # Action : Compute the median of the array.
   #
   # Return : The median of the array.
*/
{
   int    i;
   double *H_sort;

   /* Create a copy of the array whose median is to be calculated. */
   /* Necessary because the sorting function alters the array. */
   H_sort = ( double * ) malloc ( length * sizeof ( double ) );
   for ( i = 0 ; i < length ; i ++ )
      H_sort[i] = H[i];

   /* Sort the array */
```

```
   (void) bubble_sort ( H_sort, length );

   /* Return the median */
   if ( fmod ( (double) length, 2.0 ) != 0.0 )
      /* length is odd */
      return H_sort[(length-1)/2];
   else
      /* length is even */
      return (H_sort[length/2 - 1] + H_sort[length/2]) / 2.0;

   /* Free memory */
   free ( H_sort );

} /* median */


void bubble_sort ( double *H, int length )
/*
   # Arguments : 'H' is the array to be sorted. It contains
   #       'length' elements.
   #
   # Action : Sorts the array.
   #
   # Return : /
*/
{
   int i;
   int j;
   double temp;

   for ( i = length - 1 ; i > 0 ; i-- )
         for ( j = 0 ; j < i ;  j++ )
               if ( H[j] > H[j+1] )    /* compare neighboring elements */
               {
               temp   = H[j];  /* swap H[j] and H[j+1] */
               H[j]   = H[j+1];
               H[j+1] = temp;
               }

} /* bubble_sort */
```

```
/* ------------------------------------------------------
 *
 *      identifylwd.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-17
 *
 * ------------------------------------------------------ */

# ifndef IDENTTIFYLWD_H
# define IDENTTIFYLWD_H

# include "defs.h"

int identify_LWD ( char *fname_out, gps_position *gps_positions, int n_gps_positions );

void create_fine_bathymetry ( gps_position *gps_positions, \
      int n_gps_positions, \
      double *X, double *Y, double *KSI, double *H );

int median_filter_single_file ( char *prefix, \
      double *X, double *Y, \
      double *KSI, double *H_ori, int length );

int median_filter_separate_files ( char *prefix, double *X, \
      double *Y, double *KSI, double *H_ori, int length );

int spot_LWD ( char *prefix, double *X, \
      double *Y, double *H_ori, \
      double *H_flt, int length, \
      int N, int flag, double height );


# endif /* IDENTIFYLWD_H*/
```

```
/* ------------------------------------------------------
 *
 *      identifylwd.c
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-17
 *
 * ------------------------------------------------------ */

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

# include "identifylwd.h"
# include "medianfilter.h"
# include "stdev.h"
# include "defs.h"

int identify_LWD ( char *prefix, gps_position *gps_positions, int n_gps_positions )
/*
   # Arguments : prefix : prefix used to create output files.
   #             gps_positions : array of 'n_gps_positions'
   #             struct gps_position (cfr. defs.h)
   #
   # Action : Identification of LWD :
   #          1. Creation of fine bathymetry (to be filtered subsequently)
   #          2. Specification of filter length (user enters it)
   #          3. Filtering of signal.
   #          4. Comparison of original and filtered signals to spot LWD.
   #             Output results.
   #
   # Return : 0 if an error occurred, 1 otherwise.
   #          -1 if the user wants to return to the main menu.
*/
{
   /* ---------
    * Variables
    * --------- */

   double *X;  /* Array of longitudes */
   double *Y;  /* Array of latitudes */
   double *KSI;    /* Array of curvilinear coordinates */
   double *H_ori;  /* Array of depth soundings */

   int    i;
   int    length;
   int    choice;


   if ( n_gps_positions <= 0 )
   {
      printf ( "\nNo data to examine... \n" );
      printf ( "Either an error occurred while processing the raw data\n" );
      printf ( "or no data have been found. \
            Make sure to process the data first !\n" );
      printf ( "Aborting ...\n" );
      return -1;
   }


   /* Display menu */
   printf ( "\nLWD IDENTIFICATION MENU\n" );
   printf ( "-------------------------\n\n" );

   printf ( "-----------------------------------------------------------\n" );
   printf ( "|   Median filter (common file) ..................  1  |\n" );
   printf ( "|   Median filter (seperate files) ................  2  |\n" );
   printf ( "|   Standard deviation.............................  3  |\n" );
   printf ( "|                                                       |\n" );
   printf ( "|   Return to Main Menu ...........................  9  |\n" );
   printf ( "-----------------------------------------------------------\n" );
   printf ( "\n" );

   printf ( "    Selection : " );
   scanf ( "%d", &choice );

   if ( choice == 9 )   /* Return to main menu */
      return -1;

   /* ---------------
    * Allocate memory
    * --------------- */

   length = 0;
   for ( i = 0 ; i < (n_gps_positions - 1) ; i++ )
      length += gps_positions[i].n_soundings;

   X     = (double *) malloc ( length * sizeof ( double ) );
   Y     = (double *) malloc ( length * sizeof ( double ) );
   H_ori = (double *) malloc ( length * sizeof ( double ) );
   KSI   = (double *) malloc ( length * sizeof ( double ) );
```

```
   /* -----------
    * Fill arrays
    * ----------- */

   (void) create_fine_bathymetry ( gps_positions, n_gps_positions, X, Y, KSI,
               H_ori );

   switch ( choice )
   {
      case 1:
         (void) median_filter_single_file ( prefix, X, Y, KSI,
                        H_ori, length );
         break;

      case 2:
         (void) median_filter_separate_files ( prefix, X, Y, KSI,
                           H_ori, length );
         break;

      case 3:
         (void) stdev_identification ( prefix, gps_positions,
                        n_gps_positions );
         break;
   }

   free ( X );
   free ( Y );
   free ( H_ori );
   free ( KSI );

   return 1;

} /* identify_LWD */


void create_fine_bathymetry ( gps_position *gps_positions,
      int n_gps_positions, double *X,
      double *Y, double *KSI, double *H )
/*
   # Arguments : 'gps_positions' is an array of 'n_gps_positions' struct gps_position
   #             (cfr defs.h).
   #             X, Y : empty arrays to be filled with
   #             H : empty array to be filled with detailed bathymetry.
   #
   # Action : Browse the array of GPS positions and fill the other arrays. In particular,
   #          all soundings between two distinct GPS positions are given an interpolated
   #          GPS position determined by calculating the distance between the
   #          GPS position and dividing by the number of soundings.
   #
   # Return : /
*/
{
   int    i;
   int    j;
   int    index;
   double x;
   double y;
   double delta_x;
   double delta_y;
   double ksi;
   double delta_ksi;
   int    n_local;


   index = 0;
   ksi = 0.0;

   for ( i = 0 ; i < (n_gps_positions - 1) ; i++ )
   {
      /* Current GPS position */
      x = gps_positions[i].longitude;
      y = gps_positions[i].latitude;

      /* Retrieve the number of soudings for the current GPS position */
      n_local = gps_positions[i].n_soundings;

      /* Compute the increment in ksi. Note : we divide by n_local and not
         (n_local-1) because the last sounding associated with the current GPS
         position must not lie on the next one */
      delta_x   = (gps_positions[i+1].longitude - x) / n_local;
      delta_y   = (gps_positions[i+1].latitude  - y) / n_local;
      delta_ksi = gps_positions[i].distance / n_local;

      for ( j = 0 ; j < n_local ; j++ )
      {
         X[index]   = x;
         Y[index]   = y;
         H[index]   = gps_positions[i].soundings[j];
         KSI[index] = ksi;

         x   += delta_x;
         y   += delta_y;
         ksi += delta_ksi;
```

```
         index++;
      }
   }


} /* create_fine_bathymetry */


int median_filter_single_file ( char *prefix, double *X,
      double *Y, double *KSI, double *H_ori, int length )
/*
   # Arguments : prefix : prefix of output file name.
   #             X, Y : arrays of longitudes, latitudes associated with the bathymetry
   #                    (this is used to export the locations of spotted LWD)
   #      H_ori : Original bathymetry.
   #             H_flt : Filtered bathymetry.
   #           length : number of elements contained in all arrays
   #
   # Action :
   #
   # Return : 0 if an error occurred. 1 otherwise.
*/
{
   /* ---------
    * Variables
    * --------- */

   char   fname_flt[MAX_FNAME_SIZE];
   char   fname_gnuplot[MAX_FNAME_SIZE];
   FILE   *fp_flt;
   FILE   *fp_gnuplot;

   double *H_flt;
   int    N;
   int    i;
   double height;
   int    n_locations;


   H_flt = (double *) malloc ( length * sizeof ( double ) );

   /* ----------------------------------
    * Filter & identification parameters
    * ---------------------------------- */

   printf ( "\nEnter half-length of median filter : " );
   scanf  ( "%d", &N );


   printf ( "\nFiltering signal with median filter of length %d... ", 2*N+1 );
   fflush ( stdout );

   if ( median_filter ( H_ori, H_flt, length, N ) == 0 )
   {
      printf ( "An error occurred while filtering !\n" );
      return 0;
   }
   printf ( "Done\n" );


   /* ----------------------
    * Output filtered signal
    * ---------------------- */

   /* Open output file */
   strcpy ( fname_flt, prefix );
   strcat ( fname_flt, ".flt");
   fp_flt = fopen ( fname_flt, "w" );
        if ( fp_flt == NULL )
        {
                printf ( "Opening file '%s' failed in 'identify_LWD'.\n", fname_flt );
                return 0;
        }

   /* Open gnuplot data source file */
   strcpy ( fname_gnuplot, "gnuplotdata/medflt.dat" );
   fp_gnuplot = fopen ( fname_gnuplot, "w" );
   if ( fp_gnuplot == NULL )
   {
                printf ( "Opening file '%s' failed in 'identify_LWD'.\n", fname_gnuplot );
      return 0;
   }
   fprintf ( fp_gnuplot, "# %d (length of median filter)\n", 2*N+1 );


   printf ( "Exporting filtered data into '%s'... ", fname_flt );
   fflush ( stdout );
   for ( i = 0 ; i < length ; i++ )
   {
      /* Export for use under Matlab */
      fprintf ( fp_flt, "%.10f, %.10f, %.8f\n", X[i], Y[i], H_flt[i] );

      /* Export for use with Gnuplot */
      fprintf ( fp_gnuplot, "%.8f %.8f %.8f %.8f\n",
```

```
            KSI[i], H_ori[i], H_flt[i], fabs(H_ori[i] - H_flt[i]) );
   }

   printf ( "Done \n\n" );


   /* --------
    * Spot LWD
    * -------- */

   printf ( "Enter discriminative height [m] : " );
   scanf  ( "%lf", &height );

   n_locations = spot_LWD ( prefix, X, Y, H_ori, H_flt, length, N, 1, height );

   fclose ( fp_flt );
   fclose ( fp_gnuplot );
   free ( H_flt );
   return 1;

} /* median_filter_single_file */


int median_filter_separate_files ( char *prefix,
      double *X, double *Y, double *KSI, double *H_ori, int length )
/*
   # Arguments : prefix : prefix of output file name.
   #             X, Y : arrays of longitudes, latitudes associated with the bathymetry
   #                    (this is used to export the locations of spotted LWD)
   #      H_ori : Original bathymetry.
   #           length : number of elements contained in all arrays
   #
   # Action :
   #
   # Return : 0 if an error occurred. 1 otherwise.
*/
{
   /* ---------
    * Variables
    * --------- */

   char   fname_fltN[MAX_FNAME_SIZE];
   char   prefixN[MAX_FNAME_SIZE];
   FILE   *fp_fltN;

   double *H_flt;
   int    N;
   int    i;
   int    j;
   double height;
   int    common_height;
   int    n_locations;     /* Number of locations for one filtering level */
   int    n_total_locations;  /* Total number of locations for all filtering levels */

   H_flt = (double *) malloc ( length * sizeof ( double ) );

   /* ----------------------------------
    * Filter & identification parameters
    * ---------------------------------- */

   printf ( "\nEnter half-length of median filter : " );
   scanf  ( "%d", &N );

   printf ( "\nUse same height for identification of all spikes [1|0] ? " );
   scanf ( "%d", &common_height);

   if ( common_height != 0 )
   {
      printf ( "Enter discriminative height [m] : " );
      scanf  ( "%lf", &height );
   }


   /* Filtering with filters of half-lengths 1 to N */
   n_total_locations = 0;
   for ( j = 1 ; j < N+1 ; j++ )
   {

      /* Filtering with median filter of half-length j */
      printf ( "\nFiltering signal with median filter of length %d... ", 2*j+1 );
      fflush ( stdout );

      if ( median_filter ( H_ori, H_flt, length, j ) == 0 )
      {
         printf ( "An error occurred while filtering !\n" );
         return 0;
      }
      printf ( "Done\n" );


      /* ----------------------
       * Output filtered signal
       * ---------------------- */
```

```
      /* Open output file */
      sprintf ( fname_fltN, "%s_%02d.flt", prefix, j );
      fp_fltN = fopen ( fname_fltN, "w" );
      if ( fp_fltN == NULL )
      {
         printf ( "Opening file '%s' failed in 'identify_LWD'.\n", fname_fltN );
         return 0;
      }

      printf ( "Exporting filtered data into '%s'... ", fname_fltN );
      fflush ( stdout );
      for ( i = 0 ; i < length ; i++ )
            fprintf ( fp_fltN, "%.10f %.10f %.8f\n", X[i], Y[i], H_flt[i] );

      printf ( "Done \n" );
      fclose ( fp_fltN );

      /* --------
      * Spot LWD
      * -------- */

      if ( common_height == 0 )
      {
         printf ( "Enter discriminative height [m] : " );
         scanf  ( "%lf", &height );
      }

      sprintf ( prefixN, "%s_%02d", prefix, j );
      n_locations = spot_LWD ( prefixN, X, Y, H_ori, H_flt, length, j, 2, height );
      n_total_locations += n_locations;


   } /* End filtering */

   printf ( "\nTotal number of suspected locations containing LWD : %d\n\n",
         n_total_locations );

   free ( H_flt );
   return 1;

} /* median_filter_separate_files */


int spot_LWD ( char *prefix, double *X,
      double *Y, double *H_ori,
      double *H_flt, int length, int N, int flag, double height )
/*
   # Arguments : fname_lwd : name of output file.
   #             X, Y : arrays of longitudes, latitudes associated with the bathymetry
   #                    (this is used to export the locations of spotted LWD)
   #      H_ori : Original bathymetry.
   #             H_flt : Filtered bathymetry.
   #           length : number of elements contained in all arrays
   #             N : median filter half length
   #             flag : 1 to identify spikes made of 1 to N soundings
   #                    2 to identify spikes made of N soundings only
   #
   # Action : Compare original and filtered bathymetries to identify spikes that are
   #          higher (relatively to the bottom) than a height specified by the user.
   #          Locations where such spikes exist will be considered LWD-locations.
   #
   # Return : 0 if an error occurred. 1 otherwise.
*/
{
   /* ---------
    * Variables
    * --------- */
   FILE   *fp_lwd;
   char   fname_lwd[MAX_FNAME_SIZE];

   int    i;
   int    j;
   double diff;
   int    n_spotted_lwd;      /* Total number of spotted LWD */
   int    n_detected_soundings;    /* Local number of soundings shaping spike */


   /* -----------
    * Output file
    * ----------- */
   sprintf ( fname_lwd, "%s.lwd", prefix );
   fp_lwd = fopen ( fname_lwd, "w" );
        if ( fp_lwd == NULL )
        {
                printf ( "Opening file '%s' failed in 'spot_LWD'.\n", fname_lwd );
                return 0;
        }


   /* ----------------------------------------------------------
    * Examine difference between original and filtered signals
    * ----------------------------------------------------------*/

   if ( flag == 1 )   /* Detection of spikes shaped by up to N soundings */
```

```
   {
      n_spotted_lwd = 0;
      for ( i = 0 ; i < length ; i++ )
      {
         diff = H_ori[i] - H_flt[i];

         n_detected_soundings = 0;
         if ( -diff >= height )
         {
            n_spotted_lwd++;
            n_detected_soundings++;

            /* Modified on 2003.08.18 for use under fucking Windoze (for Vanketesh) */
            if ( n_detected_soundings < N+1 )
               fprintf ( fp_lwd, "%.10f, %.10f, %.8f\n", X[i], Y[i], -diff );
         }
      }
      printf ( "\n" );
      printf ( "Number of suspected locations containing LWD : %d\n\n", n_spotted_lwd );
   }
   else  /* Detection of spikes shaped by EXACTLY N soundings */
   {
      n_spotted_lwd = 0;
      n_detected_soundings = 0;
      for ( i = 0 ; i < length ; i++ )
      {
         diff = H_ori[i] - H_flt[i];

         if ( -diff >= height )
            n_detected_soundings++;
         else
         {
            if ( n_detected_soundings == N )
            {
               for ( j = i-N ; j < i ; j++ )
                  fprintf ( fp_lwd, "%.10f %.10f %d\n", X[j], Y[j], j+1 );

               n_spotted_lwd += n_detected_soundings;
            }
            n_detected_soundings = 0;
         }

      }
      printf ( "Number of suspected locations containing LWD : %d\n", n_spotted_lwd );
   }

   fclose ( fp_lwd );

   return n_spotted_lwd;

} /* spot_LWD */
```

```
/* ----------------------------------
 *
 *      graphs.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-07-23
 *
 * ---------------------------------- */

# ifndef GRAPHS_H
# define GRAPHS_H

# include "gnuplot_i.h"

int scatter_plot ( gps_position *gps_positions,
      int n_gps_positions, gnuplot_ctrl *h );

int scatter_plot_fin ( gps_position *gps_positions,
      int n_gps_positions, gnuplot_ctrl *h );

int filteredbath_plot ( gps_position *gps_positions,
      int n_gps_positions, gnuplot_ctrl *h );

int plotting_menu ( gps_position *gps_positions,
      int n_gps_positions, gnuplot_ctrl *h );

void display_plotting_menu ( );

# endif /* GRAPHS_H */
```

```
/* ----------------------------------
 *
 *      graphs.c
 *
 *      Laurent White
 *
 *      Date of creation : 2003-07-23
 *
 * ---------------------------------- */

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

# include "gnuplot_i.h"
# include "defs.h"
# include "graphs.h"


void display_plotting_menu ( )
/*
 * Arguments : /
 *
 * Action : Display plotting menu on the screen
 *
 * Return : /
 *
 * */
{

   printf ( "\nPLOTTING MENU\n" );
   printf ( "-------------\n\n" );

   printf ( "-----------------------------------------------------\n" );
   printf ( "|   Scatter plot ............................. 1   |\n" );
   printf ( "|   Fine scatter plot ........................ 2   |\n" );
   printf ( "|   Original and filtered bathymetries........... 3   |\n" );
   printf ( "|                                                     |\n" );
   printf ( "|   Return to Main Menu ...................... 9   |\n" );
   printf ( "-----------------------------------------------------\n" );
   printf ( "\n" );

} /* display_plotting_menu */


int plotting_menu ( gps_position *gps_positions, int n_gps_positions, gnuplot_ctrl *h )
/*
 * Arguments  : 'n_gps_positions' struct gps_position are passed.
 *            h is the Gnuplot handle (defined and initialized in main.c)
 *
 * Action : Display plotting menu.
 *
 * Return : -1 if user chooses to go back to main menu. 1 otherwise.
 *
 * */
{

   int choice;

   choice = -1;

   while ( 1 )
   {
      switch ( choice )
      {
         case 1:
            (void) scatter_plot ( gps_positions, n_gps_positions, h );
            break;

         case 2:
            (void) scatter_plot_fin ( gps_positions, n_gps_positions, h );
            break;

         case 3:
            (void) filteredbath_plot ( gps_positions, n_gps_positions, h );
            break;

         case 9:
            return -1;
            break;
      }

      (void) display_plotting_menu ();

      printf ( "    Selection : " );
      scanf ( "%d", &choice );
   }

   return 1;

} /* plotting_menu */


int scatter_plot ( gps_position *gps_positions, int n_gps_positions, gnuplot_ctrl *h )
/*
```

```
 * Arguments  : 'n_gps_positions' struct gps_position are passed.
 *            h is the Gnuplot handle (defined and initialized in main.c)
 *
 * Action : Scatter plot of GPS positions.
 *
 * Return : 0 if an error occurred. 1 otherwise.
 *
 * */
{

   /* Variables */
   FILE *fp_gnuplot;
   char fname_gnuplot[MAX_FNAME_SIZE];
   char fname_output[MAX_FNAME_SIZE];
   int i;
   int output_style;


   /* I/O Management */
   strcpy ( fname_gnuplot, "gnuplotdata/scatter.dat" );
   fp_gnuplot = fopen ( fname_gnuplot, "w" );
   if ( fp_gnuplot == NULL )
   {
      printf ( "An error occurred when opening file '%s' !", fname_gnuplot );
      return 0;
   }

   /* Check if there are data to plot */
   if ( n_gps_positions <= 0 )
   {
      printf ( "\nNo data to plot...\n" );
      printf ( "Either an error occurred while processing the raw data\n" );
      printf ( "or no data have been found. Make sure to process the data first !\n" );
      printf ( "Aborting ...\n" );
      return 0;
   }

   /* Reset all Gnuplot commands */
   gnuplot_cmd ( h, "reset" );

   /* Write into Gnuplot source file */
   fprintf ( fp_gnuplot, "# GNUPLOT Scatter plot\n" );
   for ( i = 0 ; i < n_gps_positions ; i++ )
      fprintf ( fp_gnuplot, "%.10f %.10f\n",
            gps_positions[i].longitude,
            gps_positions[i].latitude );

   fclose ( fp_gnuplot );

   /* Scatter plot */

   printf ( "Paper [1] or screen [2] version (Default) ? " );
   scanf ( "%d", &output_style );

   /* Depending on output style, sets different terminals */
   if ( output_style == 1 )
   {
      strcpy ( fname_output, "gnuplotfigs/scatter.eps" );
      gnuplot_cmd ( h, "set terminal postscript \"Helvetica\" 15" );
      gnuplot_cmd ( h, "set output '%s'", fname_output );
      printf ( "Output file is : '%s'\n", fname_output );
   }
   else
      gnuplot_cmd ( h, "set terminal x11" );


   /* Effective plotting takes place now ! */
   gnuplot_cmd ( h, "set nolabel" );
   gnuplot_cmd ( h, "set grid" );
   gnuplot_cmd ( h, "plot '%s' notitle with dots", fname_gnuplot );


   return 1;

} /* scatter_plot */


int scatter_plot_fin ( gps_position *gps_positions, int n_gps_positions,
      gnuplot_ctrl *h )
/*
 *  Arguments : 'n_gps_positions' struct gps_position are passed.
 *            h is the Gnuplot handle (defined and initialized in main.c).
 *
 *  Action : Scatter plot of GPS positions ('fin' stands for 'fine' -- new
 *           GPS positions have been added between existing ones by linear
 *           interpolation).
 *
 *  Return : 0 if an error occurred, 1 otherwise.
 *
 * */
{

   /* Variables */
   FILE *fp_gnuplot;
```

```
   char fname_gnuplot[MAX_FNAME_SIZE];
   char fname_output[MAX_FNAME_SIZE];
   int i;
   int j;
   double ksi;
   double x,y;
   int n_local;
   double delta_ksi;
   double delta_x,delta_y;
   int output_style;

   /* I/O Management */
   strcpy ( fname_gnuplot, "gnuplotdata/scatter_fin.dat" );
   fp_gnuplot = fopen ( fname_gnuplot, "w" );
   if ( fp_gnuplot == NULL )
   {
      printf ( "An error occurred when opening file '%s' !", fname_gnuplot );
      return 0;
   }

   /* Check if there are data to plot */
   if ( n_gps_positions <= 0 )
   {
      printf ( "\nNo data to plot...\n" );
      printf ( "Either an error occurred while processing the raw data\n" );
      printf ( "or no data have been found. Make sure to process the data first !\n" );
      printf ( "Aborting ...\n" );
      return 0;
   }

   /* Reset all Gnuplot commands */
   gnuplot_cmd ( h, "reset" );

   /* Write into Gnuplot source file */
   ksi = 0.0;

   fprintf ( fp_gnuplot, "# GNUPLOT Scatter plot (fine bathymetry)\n" );
   for ( i = 0 ; i < (n_gps_positions - 1) ; i++ )
   {
      /* Current GPS position */
      x = gps_positions[i].longitude;
      y = gps_positions[i].latitude;

      /* Retrieve the number of soudings for the current GPS position */
      n_local = gps_positions[i].n_soundings;

      /* Compute the increment in ksi. Note : we divide by n_local and not
         (n_local-1) because the last sounding associated with the current GPS
         position must not lie on the next one */
      delta_ksi = gps_positions[i].distance / n_local;
      delta_x   = (gps_positions[i+1].longitude - x) / n_local;
      delta_y   = (gps_positions[i+1].latitude  - y) / n_local;

      for ( j = 0 ; j < n_local ; j++ )
      {
         fprintf ( fp_gnuplot, "%.10f %.10f\n", x, y );

         ksi += delta_ksi;
         x   += delta_x;
         y   += delta_y;
      }
   }
   fclose ( fp_gnuplot );

   printf ( "Paper [1] or screen [2] version (Default) ? " );
   scanf ( "%d", &output_style );

   /* Depending on output style, sets different terminals */
   if ( output_style == 1 )
   {
      strcpy ( fname_output, "gnuplotfigs/scatter_fin.eps" );
      gnuplot_cmd ( h, "set terminal postscript \"Helvetica\" 15" );
      gnuplot_cmd ( h, "set output '%s'", fname_output );
      printf ( "Output file is : '%s'\n", fname_output );
   }
   else
      gnuplot_cmd ( h, "set terminal x11" );

   /* Scatter plot */
   gnuplot_cmd ( h, "set nolabel" );
   gnuplot_cmd ( h, "set grid" );
   gnuplot_cmd ( h, "plot '%s' notitle with dots", fname_gnuplot );

   return 1;


} /* scatter_plot_fin */


int filteredbath_plot ( gps_position *gps_positions, int n_gps_positions,
      gnuplot_ctrl *h )
/*
 * Arguments : 'n_gps_positions' struct gps_position are passed.
 *           h is the Gnuplot handle (defined and initialized in main.c)
```

```
 *
 * Action : Plotting of two graphs on the same page/screen : top one is the original
 *          bathymetry. Bottom one is the median filtered bathymetry.
 *
 *          IMPORTANT !! The source file used by Gnuplot is created within the function
 *          'median_filter_single_file' while exporting the data for use under Matlab.
 *
 * Return : 0 if an error occurred, 1 otherwise.
 *
 *  */
{

   /* Variables */
   char fname_output[MAX_FNAME_SIZE];
   char fname_gnuplot[MAX_FNAME_SIZE];
   FILE *fp_gnuplot;
   int output_style;
   char title1[128];
   char title2[128];
   char title3[128];
   char tmp[32];
   double x0,x1;           /* Range of x-axis */
   int filter_length;

   /* Reset all Gnuplot commands */
   gnuplot_cmd ( h, "reset" );

   /* Open Gnuplot source file to get title */
   strcpy ( fname_gnuplot, "gnuplotdata/medflt.dat" );
   fp_gnuplot = fopen ( fname_gnuplot, "r" );
   if ( fp_gnuplot == NULL )
   {
      printf ( "An error occurred when opening file '%s' !", fname_gnuplot );
      return 0;
   }
   fscanf ( fp_gnuplot, "%s %d", tmp, &filter_length );
   fclose ( fp_gnuplot );

   /* Check if there are data to plot */
   if ( n_gps_positions <= 0 )
   {
      printf ( "\nNo data to plot...\n" );
      printf ( "Either an error occurred while processing the raw data\n" );
      printf ( "or no data have been found. \
            Make sure to process the data first !\n" );
      printf ( "Aborting ...\n" );
      return 0;
   }

   /* Which output version ? */
   printf ( "Paper [1] or screen [2] version (Default) ? " );
   scanf ( "%d", &output_style );
   printf ( "\nEnter x-axis range using 'x0 x1' \
         format (for full range, enter 0 twice) : " );
   scanf ( "%lf %lf", &x0, &x1 );

   /* Depending on output style, sets different terminals */
   if ( output_style == 1 )
   {
      strcpy ( fname_output, "gnuplotfigs/medflt.eps" );
      gnuplot_cmd ( h, "set terminal postscript \"Helvetica\" 10" );
      gnuplot_cmd ( h, "set output '%s'", fname_output );
      printf ( "Output file is : '%s'\n", fname_output );
   }
   else
      gnuplot_cmd ( h, "set terminal x11" );


   /* Effective plotting takes place now ! */


   /* Titles */
   sprintf ( title1, "Median-filtered bathymetry (filter length : %d)",
         filter_length );
   sprintf ( title2, "Original bathymetry" );
   sprintf ( title3, "Absolute difference between bathymetries" );

   gnuplot_cmd ( h, "set nolabel" );
   gnuplot_cmd ( h, "set grid" );
   gnuplot_cmd ( h, "set multiplot" );
   gnuplot_cmd ( h, "set yrange [] reverse" );
   gnuplot_cmd ( h, "set size 1,0.333" );

   /* x-axis range */
   if ( ( x0 == 0.0 ) || ( x1 == 0.0 ) )
      gnuplot_cmd ( h, "set xrange []" );
   else
      gnuplot_cmd ( h, "set xrange [%f:%f]", x0, x1 );

   /* Plot 1 (top) */
   gnuplot_cmd ( h, "set origin 0,0.666" );
   gnuplot_cmd ( h, "plot 'gnuplotdata/medflt.dat' using 1:3 title \"%s\"with l",
         title1 );
```

```
    /* Plot 2 (middle) */
    gnuplot_cmd ( h, "set origin 0,0.333" );

    gnuplot_cmd ( h, "set ylabel \"Depth [m]\"" );
    gnuplot_cmd ( h, "plot 'gnuplotdata/medflt.dat' using 1:2 title \"%s\" with l",
          title2 );

    /* Plot 3 (bottom) */
    gnuplot_cmd ( h, "set origin 0,0" );
    gnuplot_cmd ( h, "set ylabel \"\"" );
    gnuplot_cmd ( h, "set xlabel \"Distance [m]\"" );
    gnuplot_cmd ( h, "set yrange [] noreverse" );

    gnuplot_cmd ( h, "plot 'gnuplotdata/medflt.dat' using 1:4 title \"%s\" with l",
          title3 );

    /* Restore normal setting */
    gnuplot_cmd ( h, "set nomultiplot" );

    return 1;

} /* filteredbath_plot */
```

```
/* ------------------------------------------------------
 *
 *      stdev.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-09-08
 *
 * ------------------------------------------------------ */

# include "defs.h"

double background_stdev ( gps_position *gps_positions, int n_gps_positions );

int compute_smooth_bathymetry ( gps_position *gps_positions, int n_gps_positions,
                      double bg_stdev, double factor );

int stdev_identification ( char *prefix,
                  gps_position *gps_positions, int n_gps_positions );
```

```
/* ------------------------------------------------------
 *
 *      stdev.c
 *
 *      Laurent White
 *
 *      Date of creation : 2003-09-08
 *
 * ------------------------------------------------------ */

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

# include "defs.h"
# include "stdev.h"

int compute_smooth_bathymetry ( gps_position *gps_positions, int n_gps_positions,
                      double bg_stdev, double factor )
/*
 * Arguments:   gps_positions : array of 'n_gps_positions'
 *              struct gps_position (cfr. defs.h)
 *
 * Action: Compute a smoothed bathymetry by discriminating those soundings
 *         that form the spike in a set of data whose standard deviation is
 *         larger than F*(background std deviation) where F is
 *         an arbitrary factor entered by the user and the background
 *         standard deviation is the RMS of all depth standard deviations.
 *
 * Return: 0 if an error occurred, 1 otherwise.
 *
 */
{
   double running_sum;
   double current_sounding;
   int i;
   int j;
   int n_local;   /* Number of soundings in local smoothed average */
   double mean;   /* Mean depth for GPS position */
   double std; /* Standard deviation for GPS position */

   running_sum = 0.0;
   n_local = 0;

   printf ( "   Computing smooth bathymetry...\n" );
   for ( i = 0 ; i < n_gps_positions ; i++ )
   {
      running_sum = 0.0;
      n_local = 0;

      /* Get mean and std deviation for current GPS position */
      mean = gps_positions[i].depth_mean;
      std = sqrt ( gps_positions[i].depth_var );

      if ( std > factor * bg_stdev )
      /* Suspected presence of LWD: calculate mean by excluding
       * those points belonging to LWD. Points related to LWD
       * are those shallower. */
         for ( j = 0 ; j < gps_positions[i].n_soundings ; j++ )
         {
            current_sounding = gps_positions[i].soundings[j];

            if ( current_sounding - mean > 0.0 )
            /* Sounding belongs to spike */
            {
               running_sum += current_sounding;
               n_local++;
            }

         } /* End browsing soundings associated
              w/ current GPS location */


      /* If no sounding is within the range, the smoothed
       * is simply taken to be the mean including all soundings */
      if ( n_local != 0 )
         gps_positions[i].depth_mean_smooth = running_sum / n_local;
      else
         gps_positions[i].depth_mean_smooth = mean;

   } /* End browsing GPS locations */

   return 1;

} /* compute_smooth_bathymetry */


int stdev_identification ( char *prefix,
                 gps_position *gps_positions, int n_gps_positions )
/*
 * Arguments : prefix : prefix of output file name.
 *             X, Y : arrays of longitudes, latitudes associated with the bathymetry
 *                    (this is used to export the locations of spotted LWD)
```

```
 *           H_ori : Original bathymetry.
 *           length : number of elements in all arrays
 *
 * Action: None
 *
 * Return: 0 if an error occurred, 1 otherwise.
 *
 */
{
   double bg_stdev;        /* Background standard deviation */
   double factor;    /* Discriminatory factor */
   int i;
   char fname_std_smooth[MAX_FNAME_SIZE];
   char fname_std_lwd[MAX_FNAME_SIZE];
   FILE *fp_std_smooth;
   FILE *fp_std_lwd;

   /* IO management */
   strcpy ( fname_std_smooth, prefix );
   strcat ( fname_std_smooth, ".std_smooth");
   fp_std_smooth = fopen ( fname_std_smooth, "w" );
        if ( fp_std_smooth == NULL )
        {
                printf ( "Opening file '%s' failed in 'identify_LWD'.\n",
          fname_std_smooth );
                return 0;
        }

   strcpy ( fname_std_lwd, prefix );
   strcat ( fname_std_lwd, ".std_lwd");
   fp_std_lwd = fopen ( fname_std_lwd, "w" );
        if ( fp_std_lwd == NULL )
        {
                printf ( "Opening file '%s' failed in 'identify_LWD'.\n",
          fname_std_lwd );
                return 0;
        }

   /* Compute background stdev (RMS of all std dev) */
   bg_stdev = background_stdev ( gps_positions, n_gps_positions );

   /* LWD identification */
   printf ( "\nLWD identification...\n" );
   printf ( "   Enter discriminatory factor: " );
   scanf ( "%lf", &factor );

   for ( i = 0 ; i < n_gps_positions ; i++ )
   {
      if ( sqrt(gps_positions[i].depth_var) > factor * bg_stdev )
         fprintf ( fp_std_lwd, "%.10f %.10f\n",
              gps_positions[i].longitude,
              gps_positions[i].latitude );
   }

   /* Compute smooth bathymetry */
   printf ( "\nCompute smooth bathymetry...\n" );
   printf ( "   Enter discriminatory factor: " );
   scanf ( "%lf", &factor );

   (void) compute_smooth_bathymetry ( gps_positions, n_gps_positions,
                  bg_stdev, factor );

   for ( i = 0 ; i < n_gps_positions ; i++ )
      fprintf ( fp_std_smooth, "%.10f %.10f %.10f\n",
           gps_positions[i].longitude,
           gps_positions[i].latitude,
           gps_positions[i].depth_mean_smooth );

   fclose (fp_std_smooth);
   fclose (fp_std_lwd);
   return 1;

} /* stdev_identification */


double background_stdev ( gps_position *gps_positions, int n_gps_positions )
/*
 * Arguments:   gps_positions : array of 'n_gps_positions'
 *              struct gps_position (cfr. defs.h)
 *
 * Action: None
 *
 * Return: The RMS (square root of the mean of the square) of
 *         all standard deviations.
 *
 */
{
   double running_sum;
   int i;

   running_sum = 0.0;

   for ( i = 0 ; i < n_gps_positions ; i++ )
      running_sum += gps_positions[i].depth_var;
```

```
   return sqrt ( running_sum / n_gps_positions  );

} /* background_stdev */
```

Bibliography

Timothy B. Abbe, David R. Montgomery, 1996. Large woody debris jams, channel hydraulics and habitat formation in large rivers. *Regulated Rivers : Research and management.* Vol. **12**, Nos. 2-3, 201-221.

N. E. Bergeron, 1996. Scale-space analysis of stream-bed roughness in coarse gravel-bed streams. *Mathematical geology.* Vol. **28**, No. 5, 537-561.

Barry J. F. Biggs, 1996. Hydraulic habitat of plants in streams. *Regulated Rivers : Research and management.* Vol. **12**, Nos 2-3, 131-144.

J. B. Butler, S. N. Lane and J. H. Chandler, 2001. Characterization of the structure of river-bed gravels using two-dimensional fractal analysis. *Mathematical geology.* Vol. **33**, No. 3, 301-330.

J. Cherry and R. L. Beschta, 1989. Coarse woody debris and channel morphology : a flume study. *Water Resources Bulletin* Vol. **25**, No. 5, 1031-1036.

J. A. Davis and L. A. Barmuta, 1989. An ecologically useful classification of mean and near-bed flows in streams and rivers. *Freshwater biology.* Vol. **21**, 271-282.

Syndi J. Dudley, J. Craig Fischenich, Steven R. Abt, 1998. Effect of woody debris entrapment on flow resistance. *Journal of the American Water Resources Association* Vol. **34**, No 5, 1189-1197.

A. Ghanem, P. Steffler, F. Hicks, C. Katopodis, 1996. Two-dimensional hydraulic simulation of physical habitat conditions in flowing streams. *Regulated Rivers : Research and management.* Vol. **12**, Nos. 2-3, 185-200.

Marc Gerhard, Michael Reich, 2000. Restoration of streams with large wood : effects of accumulated and built-in wood on channel morphology, habitat diversity and aquatic fauna. *Internationl Review of Hydrobiology.* Vol. **85**, No. 1, 123-137.

Christopher J. Gippel, 1995. Environmental hydraulics of large woody debris in streams and rivers. *Journal of Environmental Engineering.* Vol. **121**, No. 5, 388-395.

C. J. Gippel, I. C. O'Neill, B. L. Finlayson, I. Schnatz, 1996. Hydraulic guidelines for the reintroduction and management of large woody debris in lowland rivers. *Regulated Rivers : Reasearch and Management.* Vol. **12**, 223-236.

C. J. Gippel, B. L. Finlayson, I. C. O'Neill, 1996. Distribution and hydraulic significance of large woody debris in a lowland Australien river. *Hydrobiologia.* Vol. **318**, 179-194.

M. E. Harmon, J. F. Franklin, F. J. Swanson, S. V. Gregory, J. D. Lattin, N. H. Anderson, S. P. Cline, N. G. Aumen, J. R. Sedell, G. W. Lienkaemper, K. Cromack Jr. and K. W. Cummins, 1986. Ecology of coarse woody debris in temperate ecosystems. *Advances in Ecological Research.* Vol. **15**, 133-302.

Hoare, C.A.R., 2003. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM.* Vol. **4** pp. 321-322, 1961.

B. R. Hodges, 2001. Effects of large woody debris in habitat models : some thoughts on future research directions. *Civil Engineering Department, EWRE, University of Texas at Austin.* Internal note.

F. M. Henderson, 1966. Open Channel Flow. *Macmillan, New York, N Y.*

Subhash C. Jain and John F. Kennedy, 1974. The spectral evolution of sedimentary bed forms. *Joural of Fluid Mechanics.* Vol. **63**, No. 2, 301-314.

Journel, A. G. and Huijbregts, Ch. J. 1978. Mining Geostatistics. *Academic Press Inc. (London) LTD..*

Edward A. Keller and Frederick J. Swanson, 1979. Effects of large organic material on channel form and fluvial processes. *Earth Surface Processes.* Vol. **4**, No. 4, 361-380.

Joanna L. Kemp, David M. Harper, Giuseppe A. Crosa, 2000. The habitat-scale ecohydraulics of rivers. *Ecological Engineering.* Vol. **16**, 17-29.

Land and Water Australia, 2002. Fact Sheet 7 : managing woody debris in rivers. *Land & Water Australia's National Riparian Lands Research and Development Program.*

Thomas E. Lisle, 1995. Effects of coarse woody debris and its removal on a channel affected by the 1980 eruption of Mount St. Helens, Washington. *Water Resources Research.* Vol. **31**, No. 7, 1797-1808.

Michael Manga and James W. Kirchner, 2000. Stress partitioning in streams by large woody debris. *Water Resources Research.* Vol. **36**, No. 8, 2373-2379.

Martin J. Marriott, 1996. Prediction of effects of woody debris removal in flow resistance. *Journal of Hydraulic Engineering.* Vol. **122**, No. 8, 471-472.

G. W. Minshall, 1984. Aquatic insect-substratum relationships. *The Ecology of Aquatic Insects.* V. H. Resh and D. M. Rosenberg. Praeger Publishers, New York, N.Y., 358-400.

H. M. Norris, 1955. Flow in rough conduits. *Transactions of the American Society of Civil Engineers.* Vol. **120**, 373-398.

Michael Mutz, 2000. Influences of woody debris on flow patterns and channel morphology in a low energy, sand-bed stream reach. *International Review of Hydrobiology.* Vol. **85**, No. 1, 107-121.

Carl F. Nordin and James H. Algert, 1966. Spectral analysis of sand waves. *Journal of the Hydraulics Division, ASCE.* Vol. **92**, No. HY5, 95-114.

A. R. M. Nowell and M. Church, 1979. Turbulent flow in a depth-limited boundary layer. *Journal of Geophysical Research.* Vol. **84**, 4816-4824.

M. A. Oliver and R. Webster, 1986. Semi-variograms for modelling the spatial pattern of landform and soil properties. *Earth surface processes and landforms.* Vol. **11**, 491-504.

Oppenheim, A. V. and Schafer, R. W. 1999. Discrete-Time Signal Processing, second edition. *Prentice Hall Signal Processing Series.*

Osting, T., Mathews, R., Austin, B. 2003. Analysis of Instream Flows for the Sulphur River: Hydrology, Hydraulics and Fish Habitat Utilization. (Draft submitted to the US Army Corps of Engineers) *Texas Water Development Board (Surface Water Availabiliy Section).*

S. Petryk and G. Bosmajian, 1975. Analysis of flow through vegetation. *Journal of the Hydraulics Division, ASCE.* Vol. **101**, No. HY7, 871-884.

K. G. Ranga Raju, O. P. S. Rana, G. L. Asawa, A. S. N. Pillai, 1983. Rational assessment of blockage effect in channel flow past smooth circular cylinders. *Journal of Hydraulic Reasearch.* Vol. **21**, No. 4, 289-302.

Robert, A. 1988. Statistical properties of sediment bed profiles in alluvial channels. *Mathematical Geology.* Vol. **20**, 205-225.

A. Robert and K. S. Richards, 1988. On the modelling of sand bedforms using the semivariogram. *Earth Surface Processes and Landforms.* Vol. **13**, 459-473.

A. Robert, 1991. Fractal properties of simulated bed profiles in coarse-grained channels. *Mathematical geology.* Vol. **23**, No. 3, 367-382.

F. D. Shields Jr. and N. R. Nunnally, 1984. Environmental aspects of clearing and snagging. *Journal of Environmental Engineering.* Vol. **110**, No. 1, 152-165.

F. D. Shields Jr. and Christopher J. Gippel, 1995. Prediction of effects of woody debris removal on flow resistance. *Journal of Hydraulic Engineering.* Vol. **121**, No. 4, 341-354.

K. Sullivan, T. E. Lisle, C. A. Dolloff, G. E. Grant and L. Reid, 1987. Stream channels: the link between forest and fishes. *Streamside management, forestry and fishery interactions.* E. O. Salo and T. W. Cundy editions, Coll. of Forest Resour., University of Washington, Seattle, Washington, 39-97.

W. J. Young, 1992. Clarification of the criteria used to identify near-bed flow regimes. *Freshwater biology.* Vol. **28**, 383-391.

L. White, 1992. On the utilization of the semi-variogram in geostatistics. *Civil Engineering Department, EWRE, University of Texas at Austin.* Internal note.

Witkin, A. P. 1983. Scale-space filtering: Proc. Eighth Intern. Joint Conference on Artificial Intelligence (IJCAI) (Karlsruhe, Germany). Vol. **2**, 1019-1022.