River bathymetry analysis in the presence of submerged large woody debris

by

Laurent White, B.S.

Thesis

Presented to the Faculty of the Graduate School
of The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science of Engineering

The University of Texas at Austin
December 2003

River bathymetry analysis

in the presence of submerged large woody debris

APPROVED BY SUPERVISING

COMMITTEE:

Ben R. Hodges

David R. Maidment

Acknowledgments

In preparing this thesis, I have been fortunate to receive valuable assistance, suggestions, and support from my supervisor Dr. Ben R. Hodges. I would like to thank him for the invaluable experience gained during my Master's program at the University of Texas at Austin. I would also like to thank Dr. David R. Maidment for reading this thesis. My gratitude also goes to Tim Osting for his precious help and advice on this work, and to the Texas Water Developement Board for funding this project and allowing me to participate in field work. Finally, I would like to thank Alicia for proofreading the ackowledgments.

River bathymetry analysis

in the presence of submerged large woody debris

by

Laurent White

The University of Texas at Austin, 2003

SUPERVISOR: Ben R. Hodges

The frequent use of two-dimensional hydrodynamic river models 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 – LWD), simple data averaging and interpolation methods may lead to misrepresentation of the bottom bathymetry. It is necessary to distinguish in the data set what is true bathymetry from what is caused by large woody debris. Two groups of methods are investigated to serve this objective: statistical techniques and filtering techniques. In the first group, two approaches are considered: 1) a σ- discriminator method is developed and shown to effectively separate LWD from the background bathymetry, and 2) a scale-space analysis technique is applied to the same problem, but is shown to be ineffective for clearly discriminating LWD from the background bathymetry. In the second group, linear and nonlinear filters are tested. A synthesized bathymetry is used to compare relative errors associated with each method. Median filtering proves to be the best technique for removing LWD impulse spikes while leaving the background bathymetry relatively unchanged. A method of selecting

the minimum filter order based upon the physical scales of the LWD and the statistics of the data separation in the survey is proposed.

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.

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

Figure 1.1: Emergent LWD in the Sulphur River (Northeast Texas) at a low flow rate, 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).

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 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.

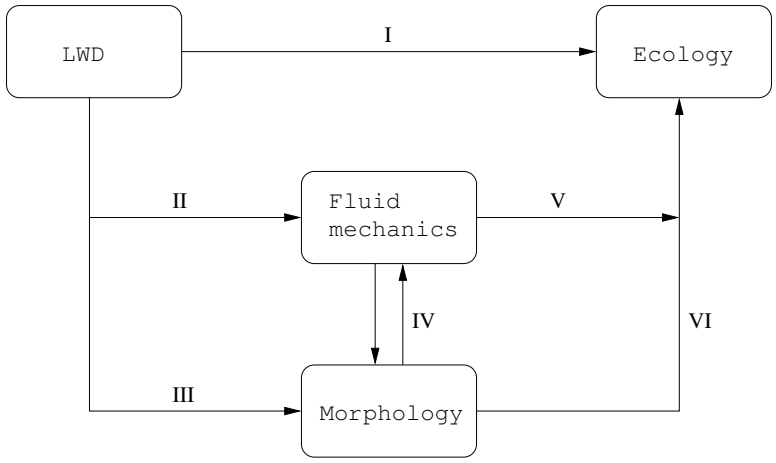

Figure 1.2: Cause-effect relationships of large woody debris within stream environments. An arrow reads *effects*

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 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 to-

tal 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

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

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:

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

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 the 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 back-water 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 a 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 is 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 generates 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

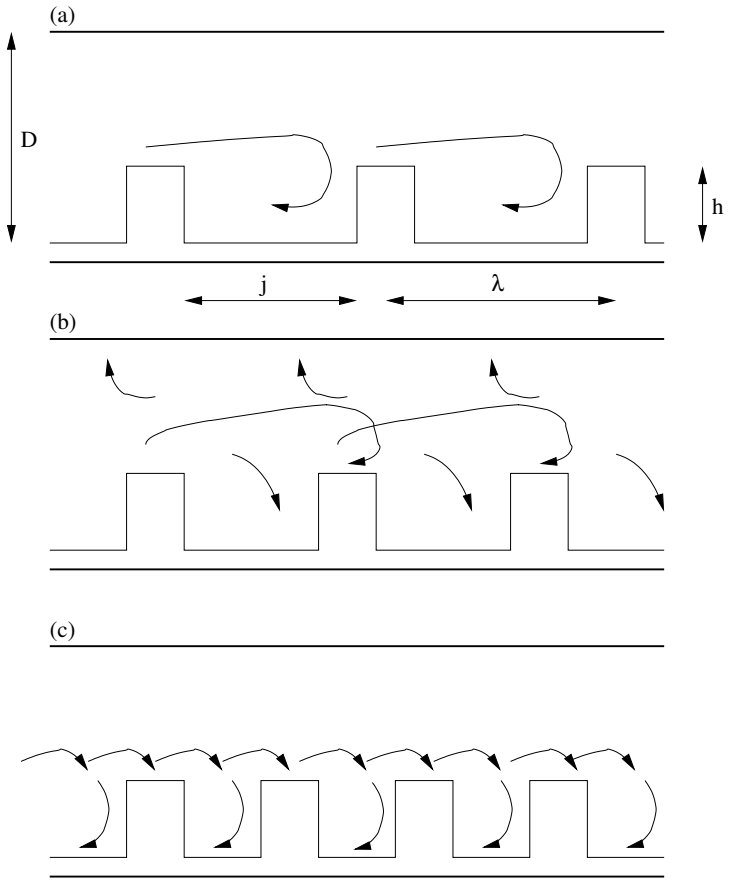

Figure 1.3: The three types of flow based upon roughness element geometry. Redrawn from Young (1992). (a) Isolated roughness flow. (b) Wake interference flow. (c) Skimming flow.

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.

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).

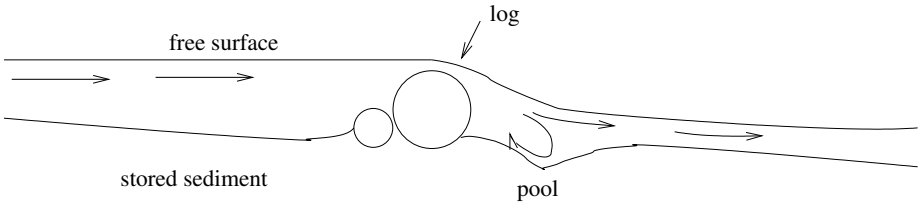

Figure 1.4: Idealized diagram showing concept of pool formation. Redrawn from Keller and Swanson (1979).

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 Thesis 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

Bathymetric field surveys

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 (Ghanem *et al.*, 1996). 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 (Ghanem *et al.*, 1996). 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 results 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 (Osting, 2003). 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.

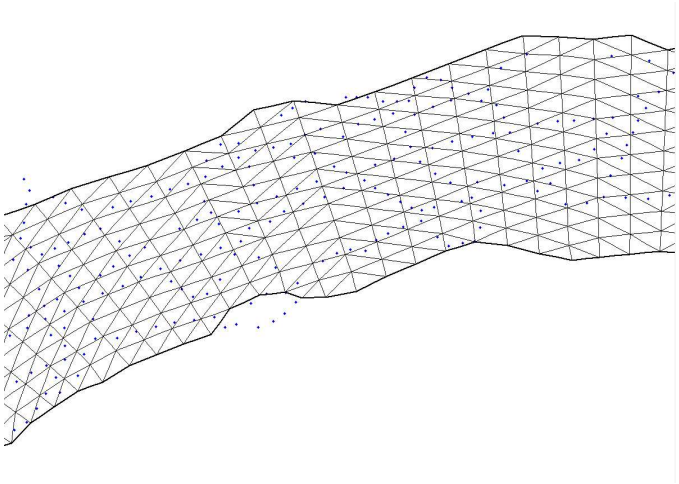

Figure 2.1: Surveyed bathymetry data points (dots) must be interpolated onto the finite element mesh.

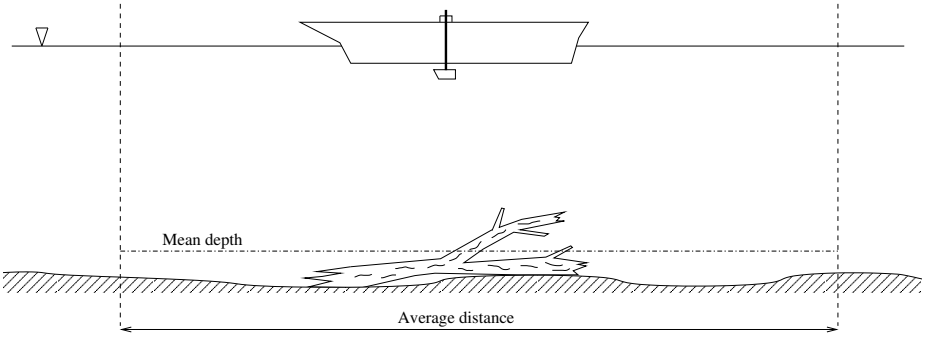

Figure 2.2: Distortion of bathymetry data due to the presence of LWD.

As a part of an aquatic habitat analysis for the Sulphur River, Texas (Osting *et 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 a 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.

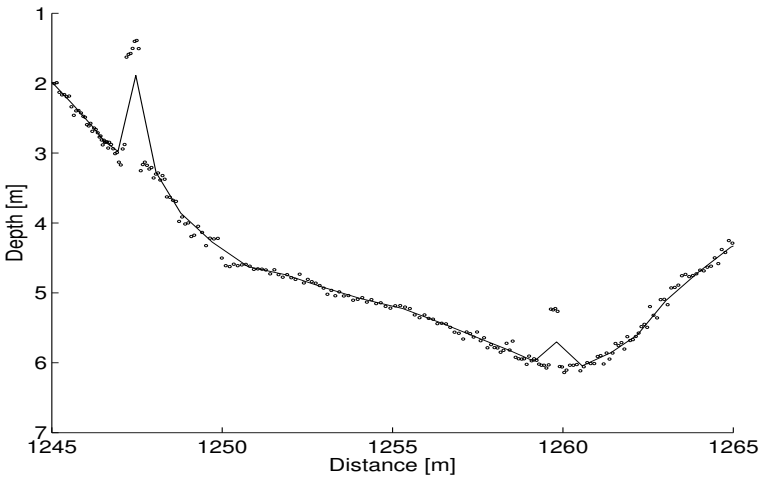

Figure 2.3: Short section of Sulphur River data set. Distributed depth measurements are represented by points while the solid line is the averaged bathymetry. Distance is measured from start of boat track in the data set.

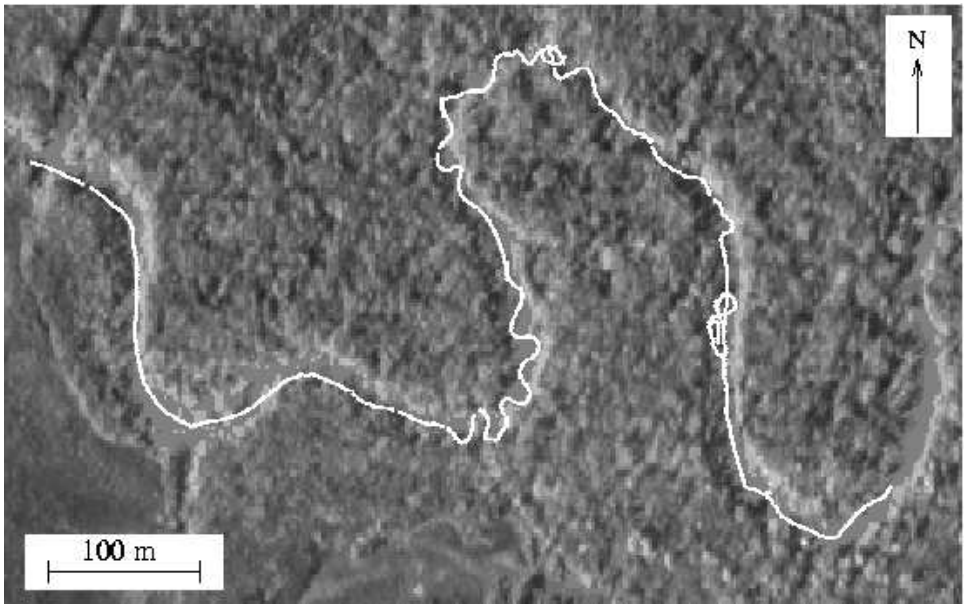

Figure 2.4: The boat track used for bathymetric analysis. This is one of several boat tracks for the Sulphur River bathymetric survey conducted in May 2001 by TWDB between (33° 18' 31.23" N, 94° 43' 37.57" W) and (33° 18' 24.18" N, 94° 43' 09.70" W).

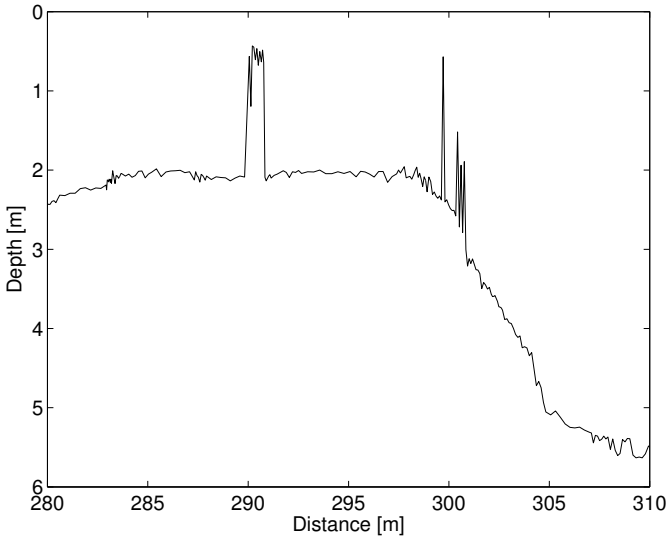

Figure 2.5: Selected section of Sulphur River bathymetry containing spikes. The average distance between depth measurements is 16 cm.

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 and Appendix B), to improve our confidence in this inference, a separate field test was conducted 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 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.

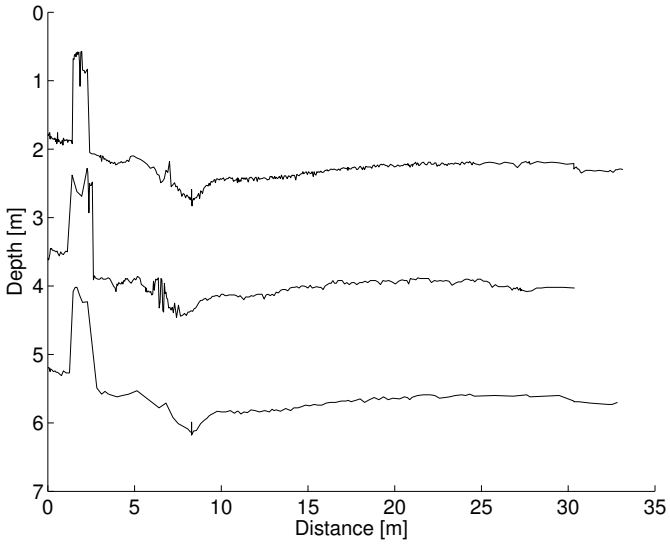

Figure 2.7: Surveyed cross-section of Guadalupe River over submerged piece of LWD (represented by the spike on the left side). Top and middle graphs show profiles obtained at different boat speeds (0.4 m s^{-1} and 0.6 m s^{-1}, respectively). Bottom graph is a decimated version of the top graph, sub-sampled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

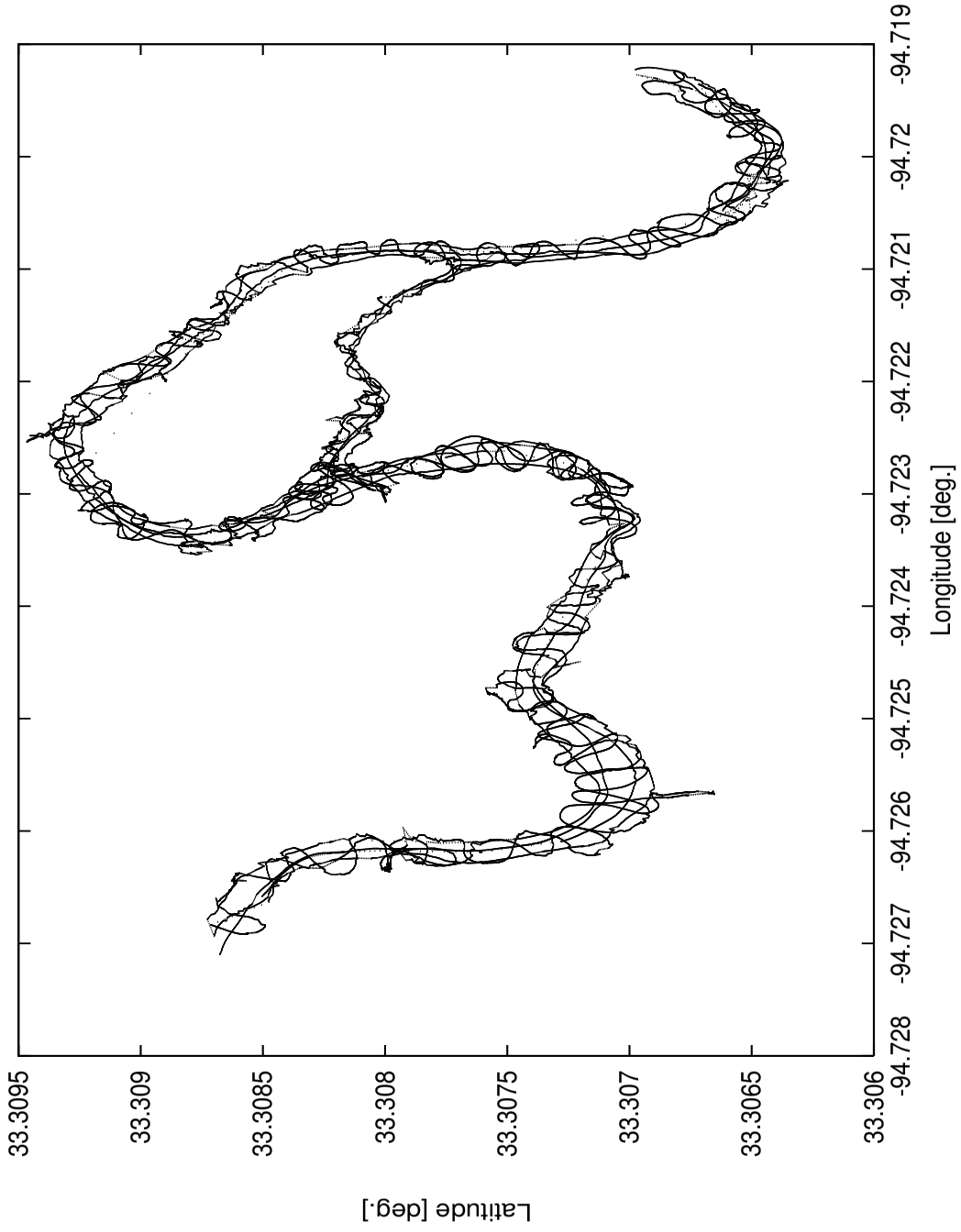

Figure 2.8: Coverage of Sulphur River by all boat tracks.

Chapter 3

Statistical techniques

In this chapter, statistical techniques are investigated as a way for identifying LWD in data sets obtained from bathymetry field surveys.

3.1 σ-discrimination of LWD

Prior surveys by TWDB used a standard approach of computing the mean depth for each distinct GPS position, providing profiles such as Figure 3.1. The bin size for each GPS position varies from six to ten depth soundings, with 86% of mean depths calculated from bins with nine data points. Binning the data leads to the disappearance of spikes that consist of only one or two depth soundings. However, broader spikes (occurring when lower survey speeds coincide with LWD) will still remain after binning. In Figure 3.1d, only spikes around distances of 1250 m and 1260 m remain after binning, while spikes at 1180 m and 1225 m disappear. When the effective survey resolution (16 cm for Sulphur River data) is of the same order as the scale of LWD, and the averaging bin (1.5 m for Sulphur River data) is larger than LWD scale, then

typical LWD will be represented by a fraction of the data in a bin. Thus, the standard deviation for a data bin provides a means of identifying the presence or absence of LWD, an approach we will call σ-discrimination.

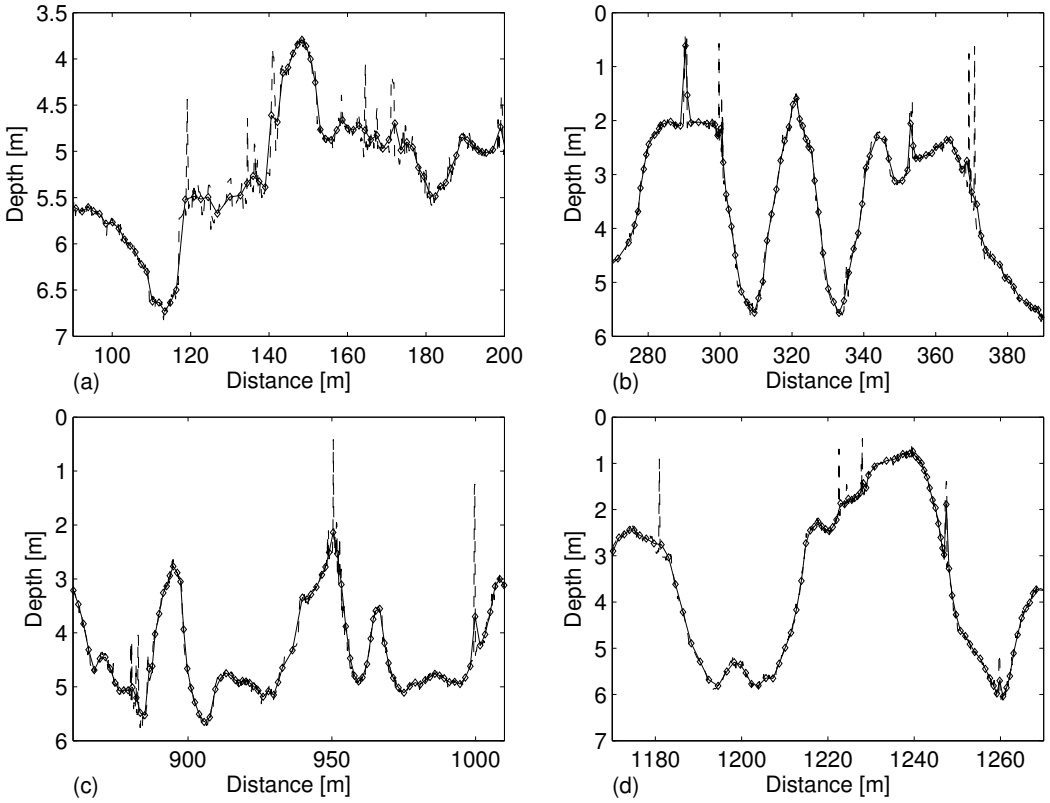

Figure 3.1: Selected sections of Sulphur River bathymetry data: diamonds represent mean depths for each distinct GPS position while the dashed line is the raw bathymetry including all depth measurements.

The bottom graph of Figure 3.3 shows that large standard deviations (σ) are associated with spikes in the raw data, which (based on the discussion above) are believed to indicate LWD.

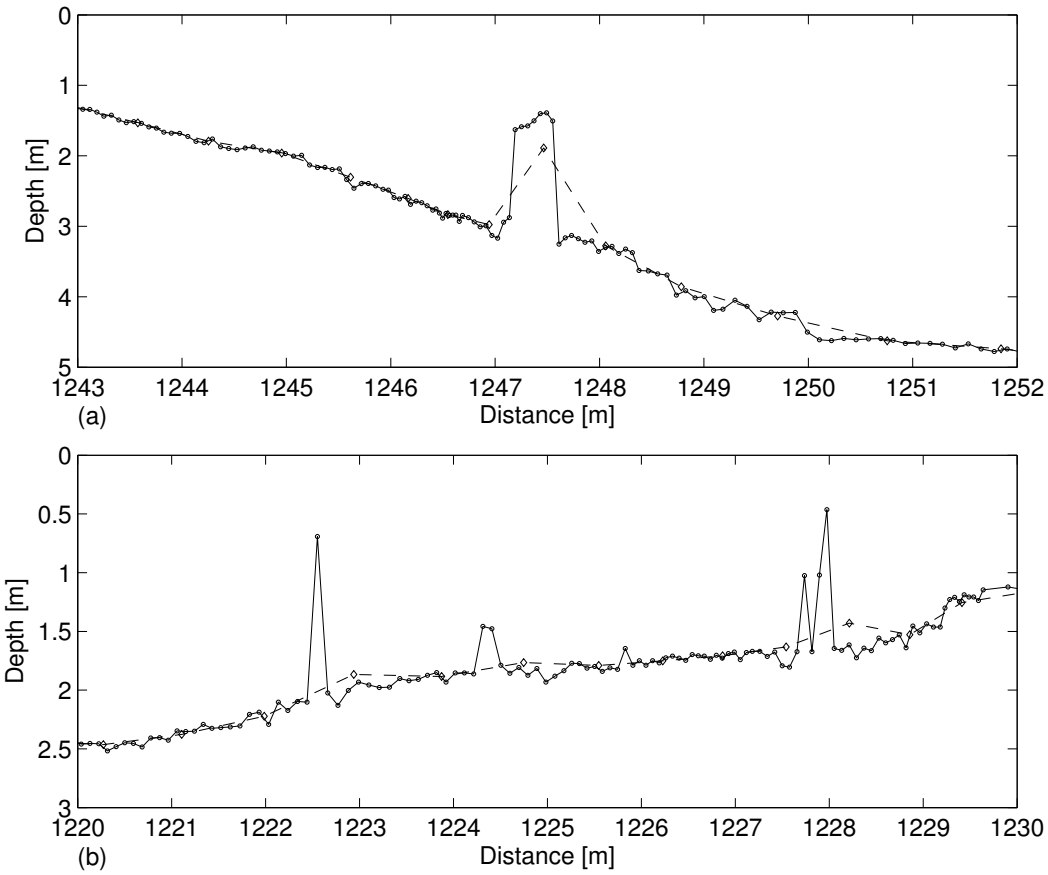

Figure 3.2: Influence of the number of depth soundings (represented by ∘) forming the spike on the calculation of the mean depth (represented by ♢ and the dashed line). (a) A spike caught by seven depth soundings leads to a high mean depth, close to the spike depth itself. (b) Spikes made of one or two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

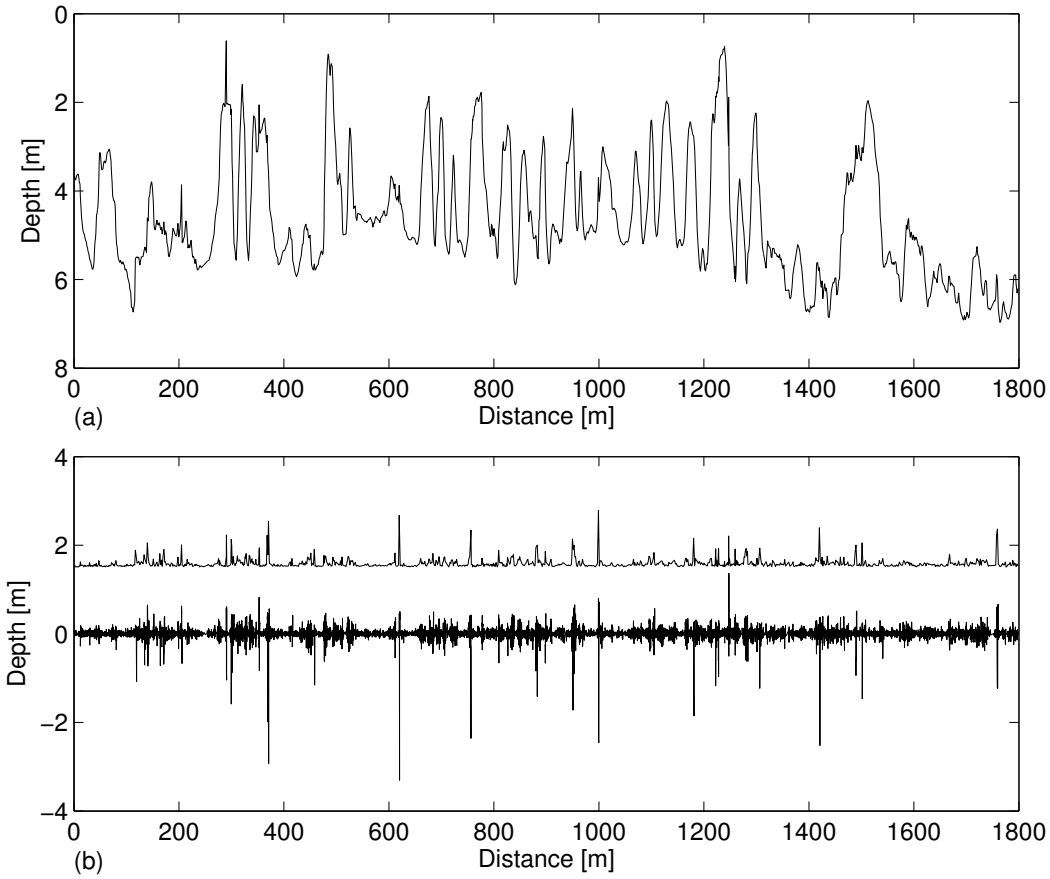

Figure 3.3: a. Mean bathymetry profile. b. The bottom line is the raw data minus the mean data. The top line is the standard deviation (relative to an arbitrary datum). Spikes in standard deviation coincide with spikes in the raw data.

As long as the majority of bins do not contain LWD, the background standard deviation (σ_B) associated with variability of the underlying bathymetry can be approximated from the RMS (root mean square) of the individual bin standard deviations (σ_i)

$$\sigma_B = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\sigma_i^2} \tag{3.1}$$

where N is the number of data bins. A bin is presumed to contain LWD if the bin standard deviation is larger than some multiple F of the background standard deviation.

$$\text{LWD discriminator} \begin{cases} \text{LWD} & \sigma_i > F\sigma_B \\ \text{No LWD} & \sigma_i \leq F\sigma_B \end{cases}$$

Selection of F is somewhat arbitrary, as the relationship between the natural roughness scales of the true bathymetry and the survey resolution will play a role in differentiating LWD from the background. However, experiments with three different values of F, shown in Figure 3.4, indicate that, at least for the present work, an appropriate F can be reasonably selected by analysis of the data set.

Increasing F leads to fewer spikes being identified as LWD. As shown in Figure 3.4b, spikes near 1220 m and 1680 m are missed when a discriminator of $3\sigma_B$ is applied. In contrast, use of a smaller F can lead to steep bathymetry slopes being misidentified as spikes. As shown in Figure 3.4f, the sloping region around 1470 m is identified as being an LWD location without any

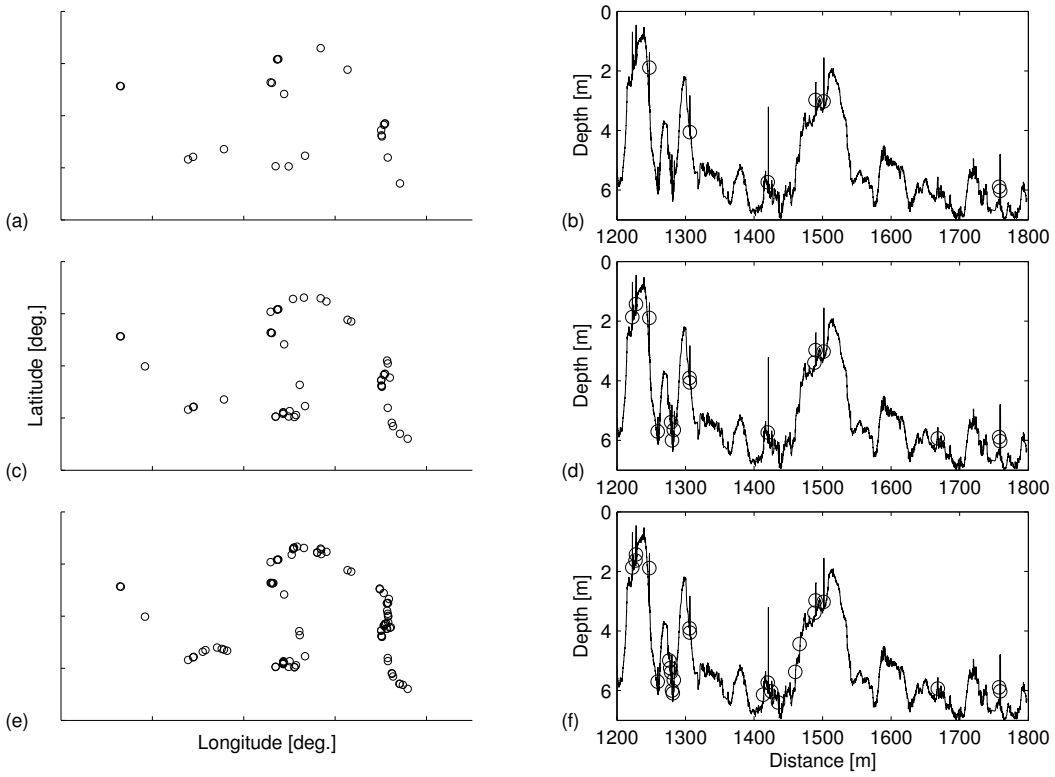

Figure 3.4: LWD identification based upon σ-discrimination. The scatter plots on the left show the suspected locations of LWD along the river. The plots on the right feature these same locations (circles) as distances along the boat track. Frames (a) and (b) use a discriminator of $3\sigma_B$. Frames (c) and (d) use $2\sigma_B$. Frames (e) and (f) use $1.5\sigma_B$. Note that the abscissa of each circle matches the GPS position of the bin while its ordinate is the bin mean bathymetry.

significant data spike when the discriminator is $1.5\sigma_B$. For the present work, a discriminator of $2\sigma_B$ appears to identify significant spikes without selecting any slope regions.

The principle drawback of the σ-discrimination approach is that a survey conducted with very slow boat speeds could produce a data set with LWD (especially large pieces or accumulations) covering multiple adjacent bins. The natural disorder of LWD accumulations may increase the standard deviation within such bins; however, a significant number of these bins could distort the calculated background standard deviation, thereby making it difficult to discriminate LWD from the natural bottom variability. Furthermore, a wide variation in the survey boat speed will lead to different areas being binned at different spatial scales. An extension of this method that might address such effects would be to bin all data within a fixed distance of each GPS position. This would ensure that all bins are averaging over the identical spatial scale. Such a technique would also be ideal for bathymetric surveys with multiple overlapping boat paths.

The σ-discrimination approach for identifying LWD locations is used to provide a "background bathymetry" that excludes the data points associated with LWD. This should be done for any bathymetry data set prior to interpolation to a coarser spatial scale for hydraulic modeling or GIS. Indeed, for GIS purposes, the LWD locations can provide an additional data layer for superposition over the background bathymetry for a more complete picture of the river characteristics. To develop a background bathymetry, we first com-

pute the mean depth in each bin based on the raw data. This raw data mean bathymetry is inherently contaminated by the presence of LWD. For bins with LWD (identified using the σ-discriminator), data points shallower than the raw data mean can be considered points where the echo sounder contacted LWD. These points are removed from the background data set. The background data set is binned to provide the estimated background bathymetry. Results for the $2\sigma_B$ discriminator are shown for three segments of the Sulphur river in Figure 3.5.

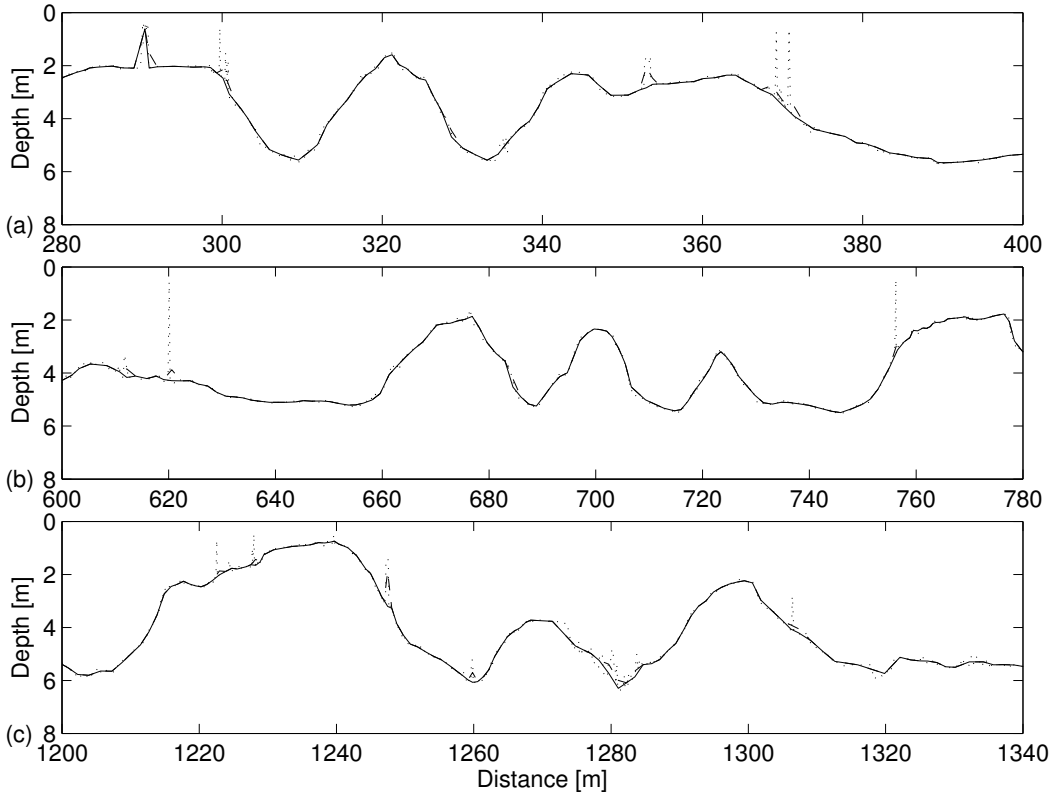

Figure 3.5: Bathymetry smoothing based upon σ-discrimination using $2\sigma_B$. The dotted line is the raw data, the dashed line is the binned raw data mean bathymetry, and the solid line is the background bathymetry.

3.2 Scale-space analysis

The scale-space filtering technique was introduced by Witkin (1983) for the analysis of digital signals. An adaptation by Bergeron (1996) provided multiscale analysis of streambed profiles, which was used to identify roughness elements at all observation scales. Scale-space filtering uses multiple successive application of a Gaussian filter (of standard deviation σ) to a data set, which can be graphed as a scale-space image (see Figure 3.6) showing successive levels of smoothing along the y-axis. Peaks and troughs of the original signal are moderated with successive applications of the smoothing filter. Plotting the physical locations of signal peaks and troughs against the smoothing level constitutes a scale-space "fingerprint" of a signal (Figure 3.7), which constitutes the multiscale description of a signal.

Each unclosed line in the fingerprint is associated with either a large-scale trough or a peak that persists despite successive smoothing. For example, the line near 310 m in Figure 3.7 shows a persistent trough, i.e. it represents the large-scale bathymetry depression between 300 and 320 m. A closed arch corresponds to the disappearance of adjacent peak-trough combinations at the associated smoothing scale. According to Bergeron (1996), smaller arches are associated with small-scale features that disappear rapidly. Bigger arches correspond to larger scale features persisting over a wider range of scales.

Discrete Gaussian filtering for scale-space analysis consists in replacing every sample by a weighted average of the bed profile over the width of the

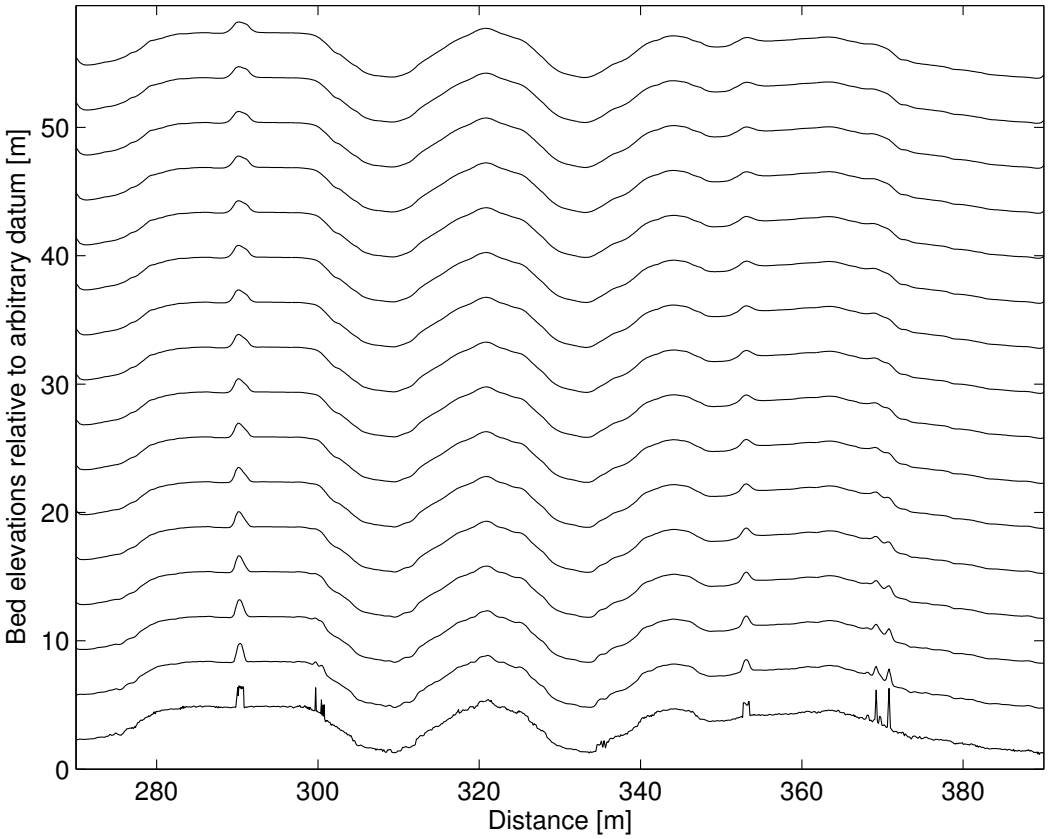

Figure 3.6: Scale-space image from 15 successive applications of a Gaussian filter with $\sigma = 20$ cm. Original bathymetry is lowermost line. Notice the smoothing (flattening and broadening) of small-scale features.

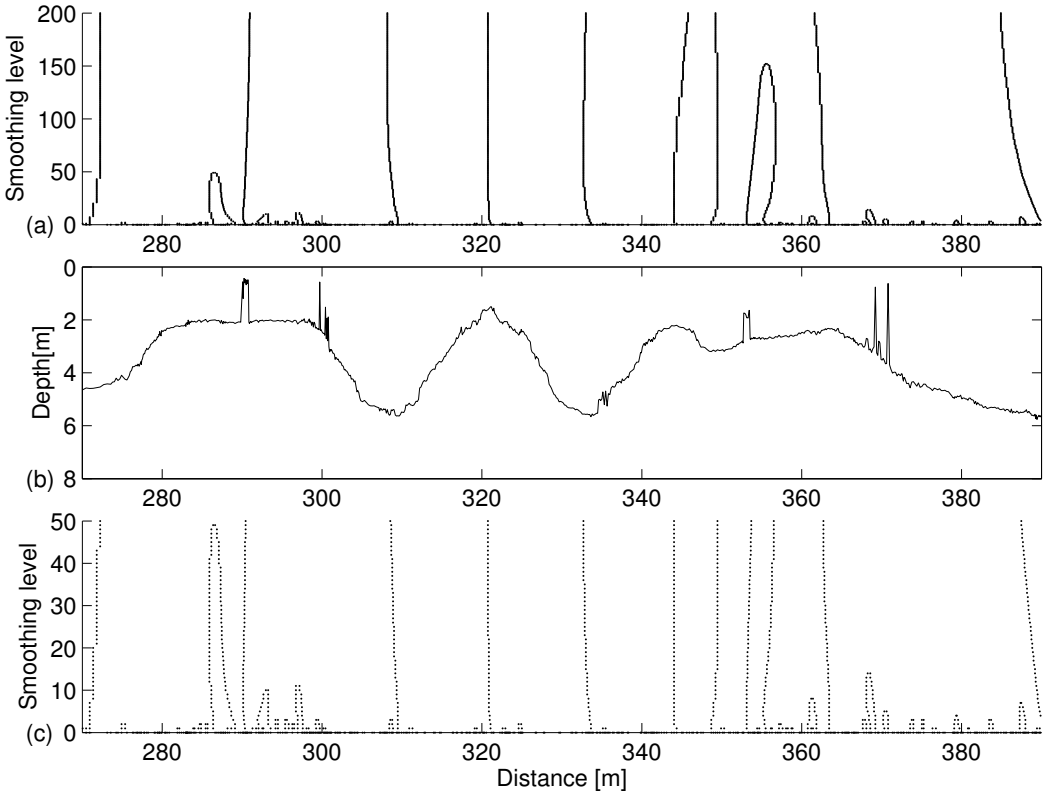

Figure 3.7: a. Fingerprint for 200 smoothing levels (Gaussian filter with $\sigma = 20$ cm applied on bathymetry shown in middle graph). b. Bathymetry. c. Highlight of the 50 first smoothing levels to make smaller arches visible.

Gaussian filter, which is centered at the sample under consideration. To implement a scale-space analysis of the Sulphur River bathymetry data set, the bathymetry is defined by the pair of sequences $\{\xi(n), x(n)\}$, where $\xi(n)$ are distances along the boat track and $x(n)$ are the depth data. As the data set is not uniformly-spaced along the boat track, the discrete Gaussian filter (with zero mean) defined at sample n takes on the following value at location k in the neighborhood of n:

$$g(k) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(\xi(k) - \xi(n))^2}{2\sigma^2}\right); \qquad n_1 < k < n_2 \tag{3.2}$$

$$= 0; \qquad \text{otherwise} \tag{3.3}$$

where $n_1 < n < n_2$ are such that the distances $|\xi(n) - \xi(n_1)|$ and $|\xi(n) - \xi(n_2)|$ are as close to the filter halfwidth as possible. That is, the sample limits n_1 and n_2 depend upon the physical distribution of the data points and are chosen so that the physical width of the filter remains approximately constant at each application. Following the recommendation of Bergeron (1996), a filter halfwidth 4σ was used. While computing a different set of $n_1 < n < n_2$ for each data point is computationally expensive, it is the only practical approach since interpolating the data to a uniform distribution for computational simplicity would distort the data spikes and invalidate the analysis. Computing the discrete filtered signal $y(n)$ from the original signal $x(n)$ is performed as

$$y(n) = \sum_{k=n_1}^{n_2} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(\xi(k) - \xi(n))^2}{2\sigma^2}\right) x(k) \tag{3.4}$$

A discrete approach to identifying lines and arches is provided below and has been implemented by the author (see Code in Appendix ??).

Arches are easily determined as follows. Once Gaussian filtering is done, producing a series of smoothing levels, each one of these is looped through to locate troughs and peaks (giving them a code, 1 for a trough and 2 for a peak), which are then recorded in another series of arrays. Each sample that is not a peak or a trough is given the code 0. By searching each smoothing level for pairs of (trough, peak), it is then determined whether the pair constitute the summit of an arch. To satisfy this property, the elements of the pair must be close enough to each other (separated by at most a number of samples fixed by the user). The "legs" of each arch are then tracked down to the first level. The height of the arch is then known in terms of the number of smoothing levels while its width is taken as being the width of the arch within the first level. The width is expressed in meters, thereby giving some length scale to the structural element that generates the arch.

The next step consists in deciding which arches are caused by LWD. This involves making assumptions as regard the likely geometry of such arches. Given a window for acceptable heights and a window for acceptable widths, an arch whose geometrical characteristics fall within these windows will be taken as being caused by LWD. Once an arch is decided to be LWD, its location must be determined. However, the width of an arch may be too wide for its precise location to be inferred. Nevertheless, since an arch is formed by the encounter of a peak leg and a trough leg, we may follow one of these legs down to the

first level, which gives a precise position along the x-axis. In a bathymetry defined in terms of depth, spikes are troughs. LWD location is then given as the trough leg of the arch.

In the context of scale-space analysis, a discrete piece of LWD in the bathymetry data should produce an arch feature. Any arch can be characterized by a width scale (W) and a smoothing scale (S). The scale W is the width of the arch in physical space at the zero smoothing level. The scale S is the smoothing level (number of applications of the Gaussian filter) at which the arch reaches a maximum. Thus, to discriminate LWD arches from the background roughness and larger scale bathymetric excursions, we need to define windows that correspond to the maximum and minimum values for each scale. Scale-space analysis does not provide any direct theory for correlating smoothing scales to physical scales, so our approach has been to examine the results of the technique for a series of different windows. As a starting point, we are interested in LWD with debris diameter scales 10 cm, so we can argue that an appropriate window minimum for W is 5 cm. The appropriate upper window limit for W is less clear, since the arch is a feature of the transition from peak to trough and can be expected to be larger than the debris feature itself. Upper window limits of 100, 150, and 200 cm for W are investigated. For the smoothing levels, we expect there to be some lower limit to the window so as not to include the background roughness of the bathymetry, and some upper limit based upon the ability of the Gaussian filter to rapidly smooth a narrow spike. Scale-space images of the bathymetry (e.g., Figure 3.6) show that data spikes

typically do not persist beyond 35 smoothing levels, so this was taken as a reasonable upper limit to the S window. For the lower S limit, we investigated smoothing levels of 5, 15 and 20.

Test cases shown in Figure 3.8 used a fixed W window of $[5, 100]$ cm while the S filter window was successively varied as $[5, 35]$, $[15, 35]$, and $[20, 35]$ smoothing levels. Although the entire data set was analyzed, for clarity only a subset of the data that is directly comparable to Figure 3.4 is shown. The smallest window, Figure 3.8a, misses most of the spikes that are likely LWD. For the $[15, 35]$ window, Figure 3.8b, additional arches are identified, but they are all spawned by small-scale features rather than high spikes. This trend continues when the window is set at $[5, 35]$ in Figure 3.8c, which adds (over the entire data set, not shown) 89 arches associated with small-scale features and 9 new arches associated with significant spikes. Decreasing the lower smoothing bound increases the false identification of LWD locations, while missing some clearly visible spikes.

A few other experiments with the same fixed W window of $[5, 100]$ cm were performed. In each case, the S window was heightened by increasing the upper smoothing level. Utilizing S windows of $[20, 40]$ and $[20, 60]$ does not improve the quality of LWD identification. In the latter case, only two new arches are identified but consist of large-scale bathymetric features.

In a second set of tests (Figure 3.9), the S filter window is fixed at $[15, 35]$ smoothing levels and the W scale window is varied as $[5, 100]$, $[5, 150]$ and

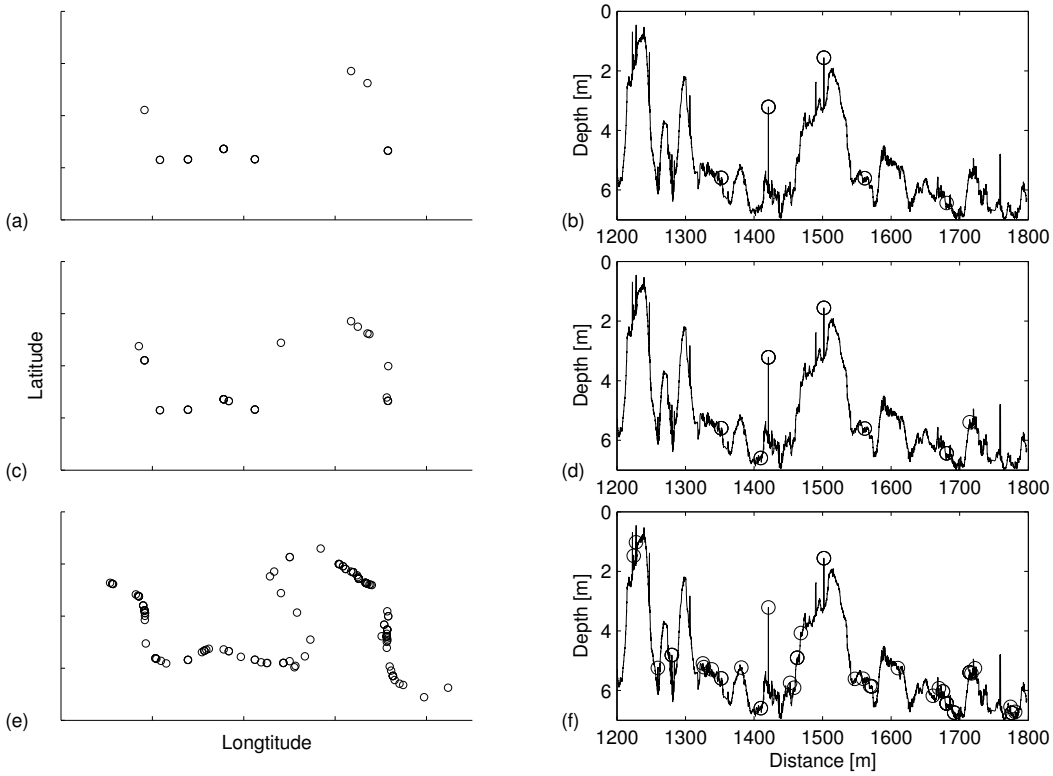

Figure 3.8: LWD Identification based upon arches geometry in the fingerprint (obtained with Gaussian filtering with $\sigma = 20$ cm). The scatter plots on the left show the suspected locations of LWD along the river. The plots on the right feature these same locations (circles) as distances along the boat track. All arches having their width between 5 cm and 100 cm are kept as LWD. Three different height windows are tested: a. Between 20 and 35 smoothing levels. b. Between 15 and 35 smoothing levels. c. Between 5 and 35 smoothing levels.

$[5, 200]$ cm. The smallest window, Figure 3.9a, is identical to Figure 3.8b. Increasing the width window to $[5, 150]$ cm in Figure 3.9b adds 31 new arches (over the entire data set) but only three are generated by spikes that could reasonably be considered LWD. Further increasing the window to $[5, 200]$ cm (Figure 3.9c) adds 40 more arches, but virtually none are associated with an LWD spike. This suggests that an overly-wide W window leads to significant false identification of LWD by including peak-trough combinations that are too large to be LWD.

Although scale-space analysis provides an interesting view of the general bathymetry structure, it does not appear to be a practical approach for identifying LWD. We have not been able to find an adequate coherence between the visually identifiable physical spikes of the non-uniform data set and the arch geometry of the fingerprint. As a result, the range of tested windows both missed data spikes that should clearly be considered LWD, and falsely identified regions of background roughness as LWD. It is not clear whether a finer sampling interval or a more uniform data set might overcome these difficulties.

3.3 Conclusions

This chapter demonstrates the use of two statistical methods for identifying submerged large woody debris in single-beam echo sounder data. The first method, σ-discrimination, is shown to be suitable for identifying likely LWD data points so that they can be separated from the original data set. This

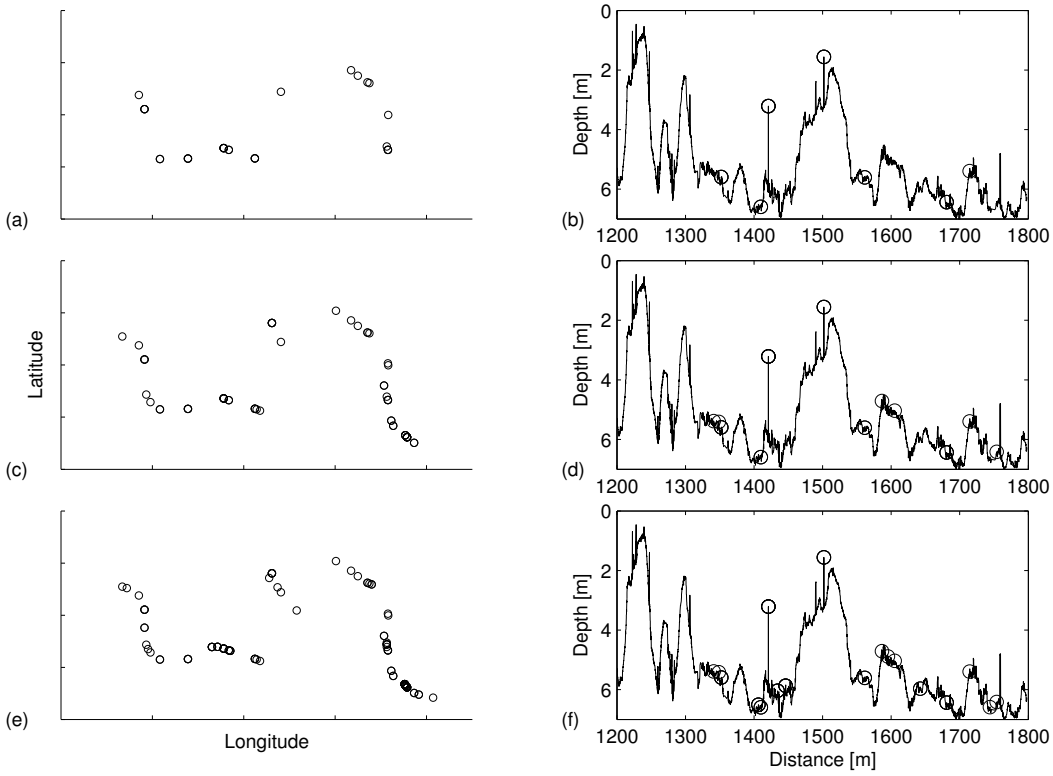

Figure 3.9: LWD Identification based upon arches geometry in the fingerprint (obtained with Gaussian filtering with $\sigma = 20$ cm). The scatter plots on the left show the suspected locations of LWD along the river. The plots on the right feature these same locations (circles) as distances along the boat track. All arches having their height between 15 and 35 smoothing levels are kept as LWD. Three different width windows are tested: a. Between 5 and 100 cm (equivalent to Figure 3.8(b) for comparison). b. Between 5 and 150 cm. c. Between 5 and 200 cm.

provides a background bathymetry that is effectively free from LWD and is appropriate for modeling or GIS purposes. Additionally, this approach provides a data set of only LWD points, which may prove useful for tracking the perennial evolution of submerged LWD fields in streams and rivers, as well as developing models which account for the physics of turbulence around LWD. The principle drawback of the σ-discrimination method is that it requires an analyst to set an appropriate standard deviation multiplier for discriminating between LWD and non-LWD data bins. As the appropriate multiplier will depend on the LWD scales and the sampling resolution, it is impossible to *a priori* set a generally applicable value. The second statistical method demonstrated, scale- space analysis, proved less successful in identifying LWD. Scale-space analysis for identifying LWD requires setting upper and lower windowing limits on the "fingerprint" width and smoothing height of "arches" associated with LWD. In the present work, we were unable to find a suitable set of windows that identified the majority of LWD without also providing a significant number of false positives.

Chapter 4

Filtering techniques

Linear and nonlinear filters are examined through their application to a synthesized bathymetry. Their relative efficacy in spikes removal is evaluated.

4.1 Methodology

An artificial bathymetry has been synthesized to examine the performance of linear and nonlinear filtering techniques in a controlled fashion. Field data from the Sulphur River is used to further analyze the capabilities of the more successful approach (nonlinear filtering). The key difficulty for any filtering technique (which can be tested on a synthetic bathymetry) is differentiating between LWD data spikes and abrupt transitions that are part of large-scale features in the natural bathymetry. The synthetic bathymetry data (Figure 4.2) is 300 data points, which includes two sharp edges surrounding a 1 m high and 15 m long upward rise. The left edge of the rise has a slope of 1:2.5 while the right edge has a shallower slope of 1:5. Three spikes (representing LWD) are superposed on the bathymetry: a narrow spike (A) on top of the

bathymetry rise, a second narrow spike (B) on one slope, and a broader spike (C) on a flat part of the bathymetry. Each spike has a 1 m amplitude. The two narrow spikes each contain a single data point, while the broader spike has six data points. The distance between successive samples is 16 cm, which is the average distance between depth measurements in the TWDB Sulphur River survey. Thus, the width of the narrow spikes is of order 10 cm, similar to the data signal returned when a survey crosses a piece of LWD at a right angle. The width of the broad spike is of order 100 cm, similar to the data signal for a piece of LWD crossed at an oblique angle by a survey. A noise signal was superposed on the bathymetry to represent natural variability in a river bottom and oscillations that may be caused by motion of a survey boat. The noise signal is developed by a random number generator with a uniform distribution and maximum amplitude of 0.025 m. This is consistent with the noise signal computed from a fine-scale survey over a cross-section of the Guadalupe River conducted in April, 2003 using the same equipment as the TWDB Sulphur River survey (see previous Chapter). The objective of filtering the synthetic bathymetry is to remove the three spikes while retaining a good representation of the underlying bathymetry.

4.1.1 Linear filtering

Linear filters are applied by convolving the filter kernel with the signal to be filtered. Convolution can be efficiently performed by using a Fast Fourier Transform (FFT) algorithm to compute the Discrete Fourier Transform (DFT). In practice, DFT's of the filter kernel and the signal are mul-

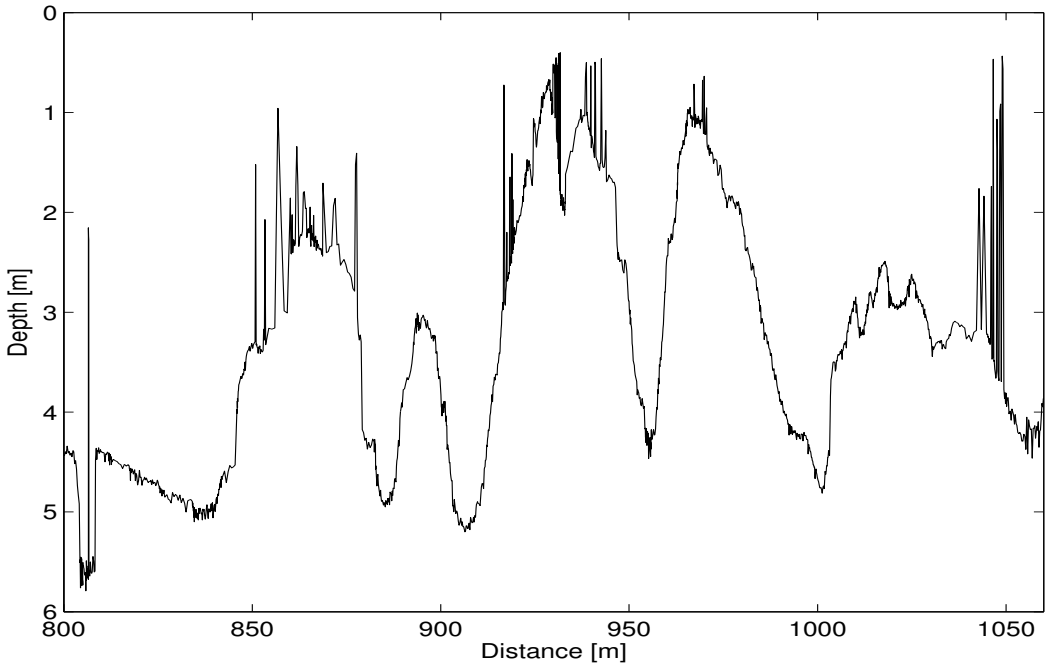

Figure 4.1: Section of Sulphur River bathymetry featuring severe spikes.

tiplied, and then the inverse DFT of their product provides the filtered signal. A low-pass filter is designed to remove the high wave-number data (i.e., the LWD spikes with short wavelengths) from a digital signal, leaving only the low wave-number (i.e., long wavelength) signal that should represent the bathymetry without LWD. In applying a linear low-pass filter, the cutoff wave number must be *a priori* defined. For a method to work mechanistically, the cutoff wave number should have a defined relationship to the physical properties of the LWD. The wave number (k) is related to the wavelength (λ) by $k = 2\pi/\lambda$, so a low-pass wave number filter is also a high-pass wavelength filter. Linear filtering decomposes a signal into a linear sum of sine waves, so that any spike or sharp transition must be composed of wave lengths substantially smaller than the spike. It follows that the high-pass cutoff wavelength

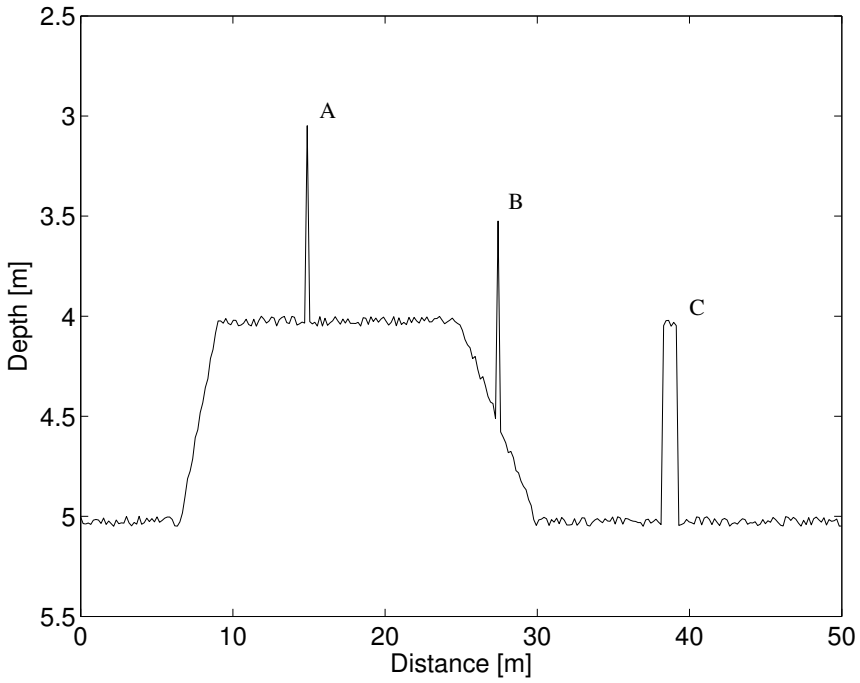

Figure 4.2: Synthesized bathymetry providing a benchmark for filters. Both narrow spikes are made up of one data point while the broader spike contains six data points.

should be an order of magnitude smaller than smallest LWD physical length scale ($\mathcal{L}$). Thus, for the synthetic bathymetry LWD spikes with $\mathcal{L} \approx 10$ cm and $\mathcal{L} \approx 100$ cm, the cutoff wavelength should be 1 cm or less, leading to a requirement for low-pass cutoff wave numbers greater than 6.3 cm^{-1}. The performance of a representative set of cutoff wave numbers $[7.5, 15, 30, 45]$ cm^{-1} are analyzed below to illustrate how the predicted background bathymetry and spike removal depend on the wave number selection. Smaller cutoff wave numbers were tested (not shown), but did not significantly change the filtered signal.

Linear filters fall into two categories: 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. The faster IIR filters are only conditionally stable – feedback is used – so particular care must be taken in their design (Oppenheim and Schafer, 1999). The poles of the rational transfer function characterizing IIR filters must lie inside the unit circle in the z-plane. Since no feedback is used for FIR filters, their transfer functions do not have poles and stability is not an issue. In the following, the results of an FIR filter of order 50 and an IIR filter of order 10 are presented. Filtering has been performed using the Matlab$^{\text{TM}}$ Signal Processing Toolbox, which limits the FIR filter order to one-third of the signal length. This limit allows a maximum FIR order of 100 for the synthetic bathymetry. However, tests conducted on the synthetic bathymetry with FIR filter orders higher than 50 produced poorer results (not shown). In particular,

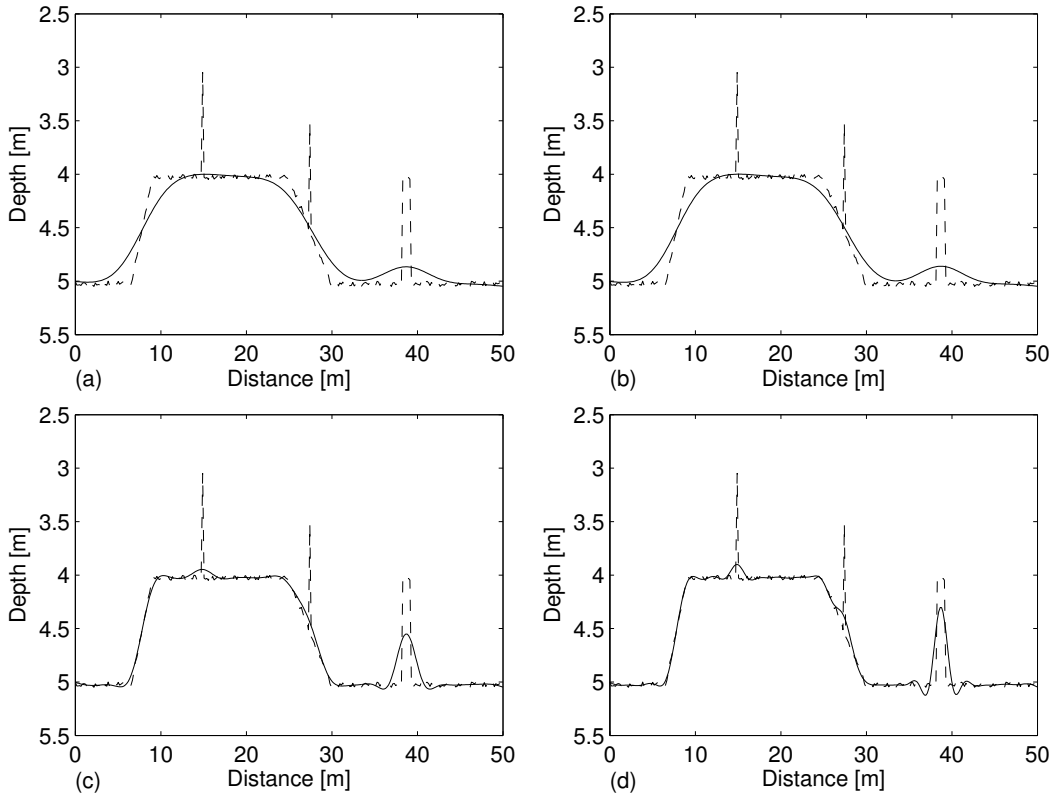

Figure 4.3: The dashed and solid lines represent the original and filtered signals, respectively. An FIR filter of order 50 has been used. Cutoff wave-numbers are: (a) 7.5 cm^{-1}, (b) 15 cm^{-1}, (c) 30 cm^{-1} and (d) 45 cm^{-1}.

increasing the order beyond 50 does not improve spike removal, but induces additional oscillations in the neighborhood of the spike. For IIR filters, beyond an order of 10 the filter becomes unstable and cannot be applied. While linear filters are well-known and computationally efficient, their use tends to remove sharp details that may not be either noise or LWD. The results of our analysis show that distortion of the underlying bathymetry makes linear filters less suitable than nonlinear filters for LWD analysis.

A Hamming-windowed FIR linear filter of order 50 (with four different cutoff wave numbers) has been applied to the synthetic bathymetry. Windowing truncates the ideal low-pass filter (for an infinite length signal) in the physical-space domain. Figure 4.3a shows that the lowest wave number cutoff will remove both narrow spikes and excessively smoothes the sharp edges in the bathymetry. Decreasing the cutoff wave number below the smallest value (used in Figure 4.3a) does not significantly change the results: the narrow spikes are removed and the broader spike remains a smoothed bump. A significant problem with the low wave number cutoffs in Figure 4.3a and 4.3b is distortion of the large-scale slopes. For both cases, the left and right edge slopes have been reduced from 1:2.5 and 1.5 to approximately 1:6.5 and 1:8, respectively. For a higher wave number cutoff, Figure 4.3c, the slopes are preserved but transitional corners are rounded by the filter. At the highest wave number cutoff (Figure 4.3d), the correct slopes are preserved, but the majority of the broader spike and smaller portion of the narrower spikes still remain.

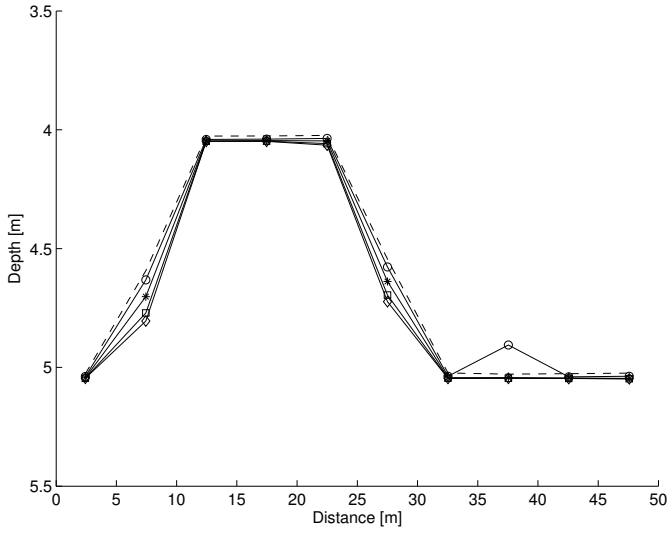

Figure 4.4: Comparison of digitized spike-free synthetic bathymetry (dashed line) with digitized filtered bathymetries. Digitized filtered bathymetries are related to Figure 4.3 as follows: Graph a: $\circ$, Graph b: $\star$, Graph c: $\square$, Graph d: $\lozenge$.

River bathymetry collected at a fine scale may be applied at coarser scales for use in developing TINs for GIS or modeling. Thus, the performance of a filtering method is better analyzed by how well a coarse digitization of filtered results compares to a similar digitization of the synthetic bathymetry without the spikes. Figure 4.4 shows the filtered bathymetries from Figure 4.3 where the bathymetry at 5 m spacings are computed from the mean of data binned from the surrounding 5 m interval. The relative error (ϵ) between the filtered and the spike-free original bathymetry is computed as

$$\epsilon = \frac{\left[\sum (h_f - h_0)^2\right]^{1/2}}{\left[\sum (h_s - h_0)^2\right]^{1/2}} \tag{4.1}$$

where h_f and h_0 are the mean digitized values of the filtered digital bathymetry and the original spike-free bathymetry on 5 m intervals, and h_s is the mean dig-

itized values of the synthetic bathymetry including the spikes. Thus, the relative error is a measure of how well the filtering reduces the error in the digitized signal below the error resulting from digitizing without filtering. The relative errors obtained for FIR filtering are displayed in Table 4.1 along with errors for IIR filtering and nonlinear filters (discussed below). Digitized bathymetries using 2 m spacings and 3.5 m spacings have also been computed. The relative errors obtained (not shown) are not improved upon that for the 5 m spacings.

We tested three types of IIR filters: Butterworth, Type 1 Chebyshev, and elliptic. All the filters gave similar results, so only the Butterworth filter results are presented below. The same cutoff wave numbers used in the FIR results were employed with IIR filters to obtain Figure 4.5. While the tested FIR filter was order 50, the maximum IIR filter order was 10 to prevent instabilities for low wave-number cutoffs. It can be seen that the lowest cutoff wave number (Figure 4.5a) produces a similar slope distortion to the FIR results in Figure 4.3a, and causes additional distortion of the flat bathymetry section with the broad spike. At the cutoff wave number of 15 cm^{-1}, the slope error is less severe than for the equivalent FIR (Figure 4.3b), but additional oscillations appear near the sharp transitions. At higher cutoff wave numbers (Figure 4.5c,d) the IIR results are not substantially different from FIR results. The mean error for the IIR results digitized at 5 m intervals are computed in the same manner as for the FIR, and are provided in Table 4.1. It can be seen that the relative errors are not substantially different. It should be pointed out that a relative error exceeding 100% indicates that the original digital bathymetry

(with spikes) is actually closer to the original digital spike-free bathymetry than the filtered digitial bathymetry.

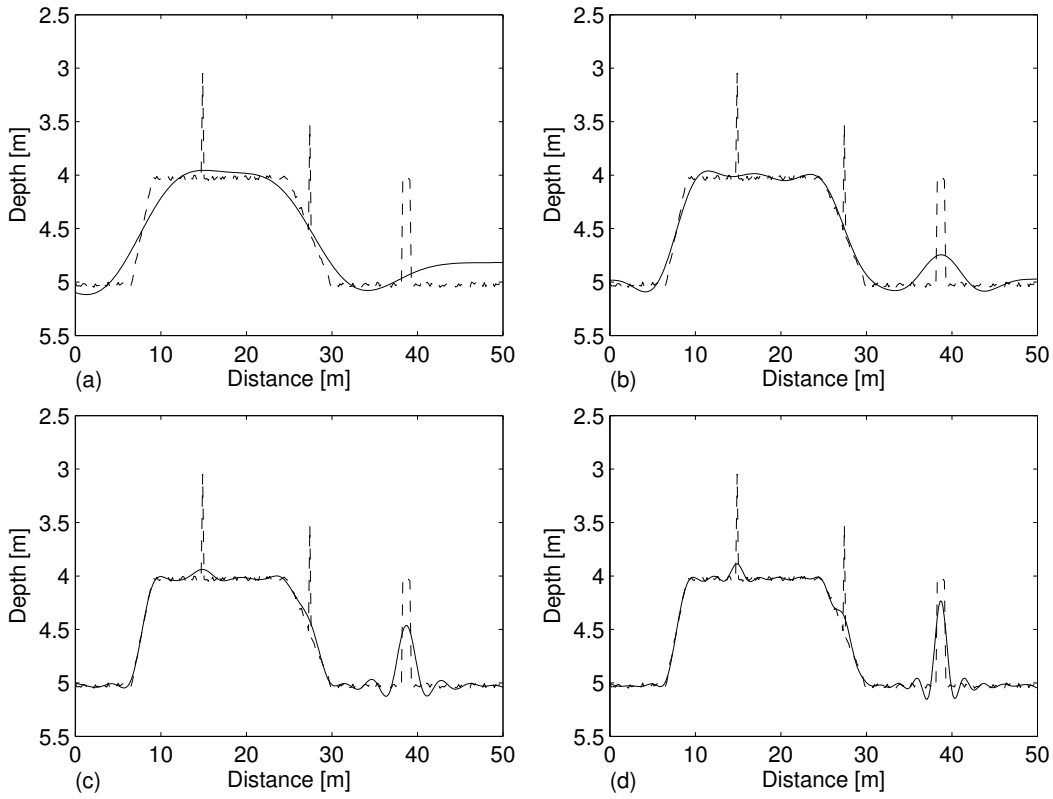

Figure 4.5: The dashed and solid lines represent the original and filtered signals, respectively. A Butterworth filter of order 10 has been used. Cutoff wave-numbers are: (a) 7.5 cm^{-1}, (b) 15 cm^{-1}, (c) 30 cm^{-1} and (d) 45 cm^{-1}.

4.1.2 Nonlinear filtering

The spikes in the artificial bathymetry (Figure 4.2) are typical *impulse noise* signals, whose removal via linear filtering will distort any sharp edges of larger-scale features. Nonlinear filtering techniques, however, may remove impulse noise without disturbing edges (Justusson, 1981). Nonlinear filters may be

more computationally demanding than linear filters, depending on the nonlinear filter order and the length of the data set. Nonlinear filters require a selection operation for each data point, resulting in $\mathcal{O}(LK)$ operations, where K is the length of the window and L is the length of the data set. In contrast, the linear filtering FFTs require $\mathcal{O}(L \log_2 L)$ operations. Nonlinear filters are discussed in terms of their order N, where the window length is $K = 2N + 1$. It follows that nonlinear filtering will be more efficient than linear filtering only when $N < 0.5 \log_2 L$. We tested nonlinear erosion and median filters in the present study. The median filter is widely used in image processing to remove impulse or "salt-and-pepper" noise without blurring sharp edges in an image (e.g., Justusson (1981), King *et al* (1989), Dougherty and Astola (1999)). Image processing also makes use of erosion filtering to dim images by thinning out lighter elements (i.e. signal peaks).

Both erosion and median nonlinear filters use order statistics. For each point $\tilde{x}$ in the sequence $x(n)$, a filter window of length $K = 2N + 1$ is formed by taking those N points preceding $\tilde{x}$ and those N points succeeding it. This set is then ordered from the smallest to largest value. Erosion filtering replaces the value at point $\tilde{x}$ with the minimum of the ordered set surrounding the point. Median filtering replaces the value at $\tilde{x}$ with the median of the ordered set. An example of median filtering with $N = 2$ for a small spike on a slope is shown in Figure 4.6. The erosion filter operates similarly, except that it uses the minimum of the order statistics instead of the median, and results in filtered values of $y(7) = 2.5$ and $y(8) = 3.0$ for the example shown. Data

points within N points of the start or end of the signal do not have $2N + 1$ data points in filter window, which is commonly handled by appending the signal with copies of the start and end values (Justusson, 1981).

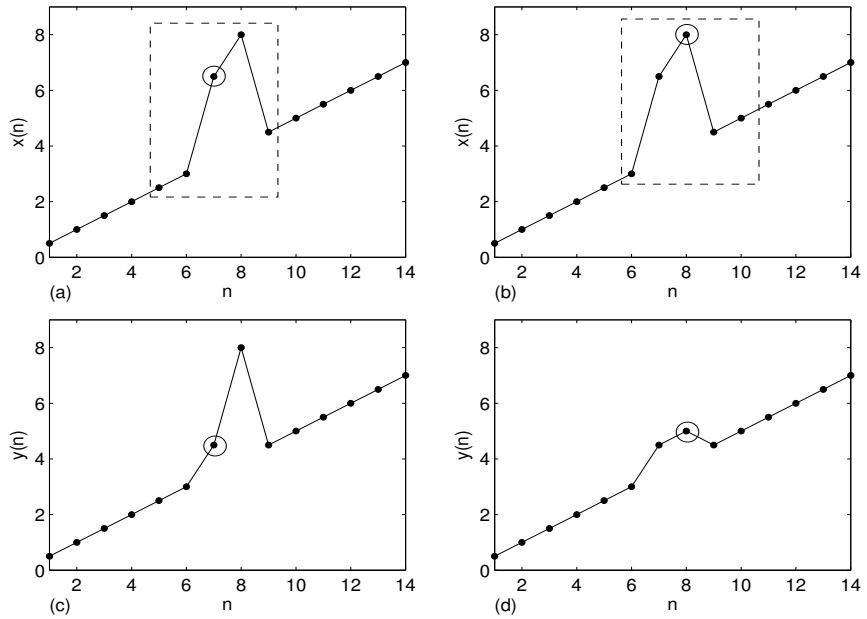

Figure 4.6: Illustration of median filtering for $N = 2$. $x(n)$ and $y(n)$ are the original and filtered sequences, respectively. The dashed box contains the five n samples to be algebraically ordered by $x(n)$ and $y(n)$ values. Plots (a) and (c) show the filtering for $n = 7$ (circled), where the median of the boxed values provides $y(7) = 4.5$. Plots (b) and (d) show the filtering for $n = 8$ (circled), where the median of the boxed values provides $y(8) = 5.0$.

Erosion filtering selects the minimum of an ordered set, a bias which would *increase* an LWD signal if applied to a set of depth measurements (since the LWD causes a decrease in depth). Thus, for an erosion filter the bathymetry data set must be redefined as the height above a datum or, equivalently, the erosion filter can be redefined as selecting the maximum of the ordered depth data in a window. The effect of erosion filtering on the synthetic bathymetry

is shown in Figure 4.7, where four different filter lengths have been used. The $N = 1$ erosion filter is able to eradicate both narrow spikes (Figure 4.7a), but cannot remove the broader spike (although it does reduce the spike width). A $N = 3$ erosion filter is required to remove the broad spike (Figure 4.7b). Although the large N erosion filters are able to preserve the sharp transitions and slopes of the bathymetric rise, it can be seen that the overall width of the rise is successively reduced as N increases (i.e., the rise is eroded). In contrast, results of median filtering (Figure 4.8) show that the higher-order median filters remove the spikes without significantly changing the bathymetric rise. However, whereas an $N = 3$ erosion filter removed all the spikes, an $N = 6$ median filter is required to achieve the same result (Figure 4.8b). Relative errors between the digitized filtered bathymetries and the synthetic spike-free digitized bathymetry (using Eq. 4.1) are displayed in Table 4.1.

4.2 Discussion

From Table 4.1 as well as Figures 4.3, 4.5, 4.7, 4.8 and 4.9, it is evident that linear filtering is largely outperformed by nonlinear filtering, with the median filter providing the best removal of LWD while retaining the truest representation of the background bathymetry. A key difficulty of linear filtering is the tradeoff between spike removal and distortion of the background bathymetry. With lower cutoff wave numbers, linear filtering can guarantee spike removal, but will significantly distort steep slopes in the background. In contrast, higher cutoff wave numbers allow the slopes to be retained, but have poor removal of larger spikes that may result from a survey crossing LWD at an oblique angle.

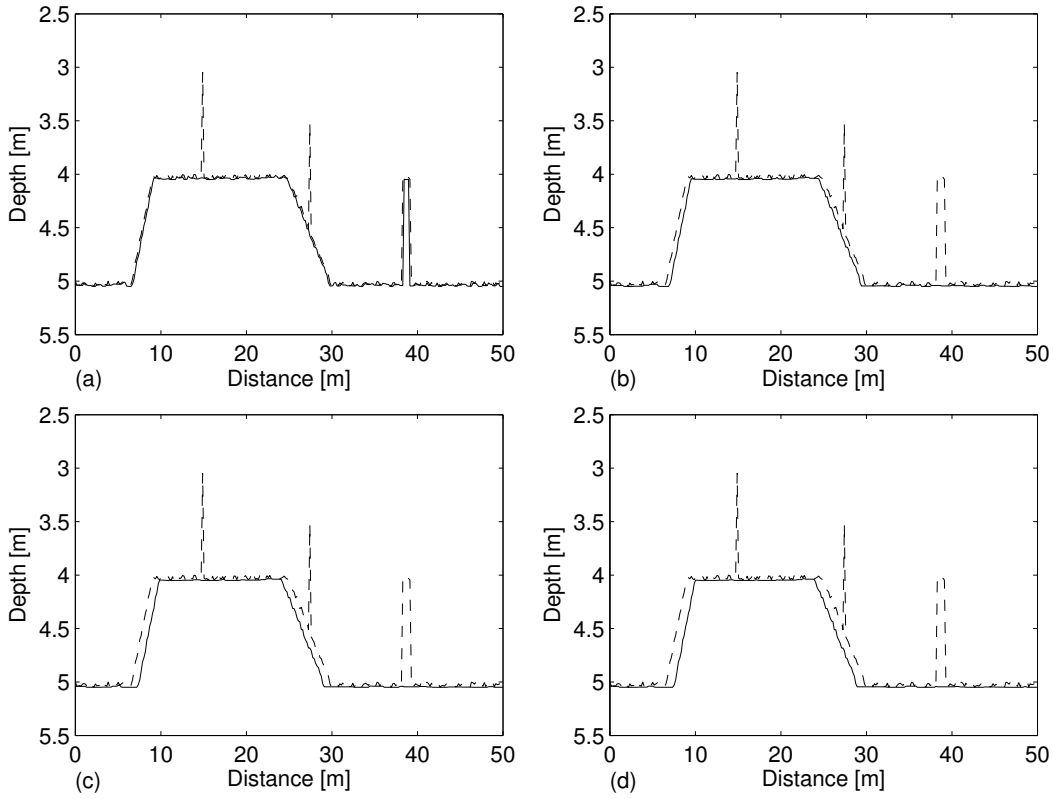

Figure 4.7: The dashed and solid lines represent the original and erosion-filtered signals, respectively. The following filter lengths have been used: (a) 3 ($N = 1$), (b) 7 ($N = 3$), (c) 11 ($N = 5$) and (d) 13 ($N = 6$).

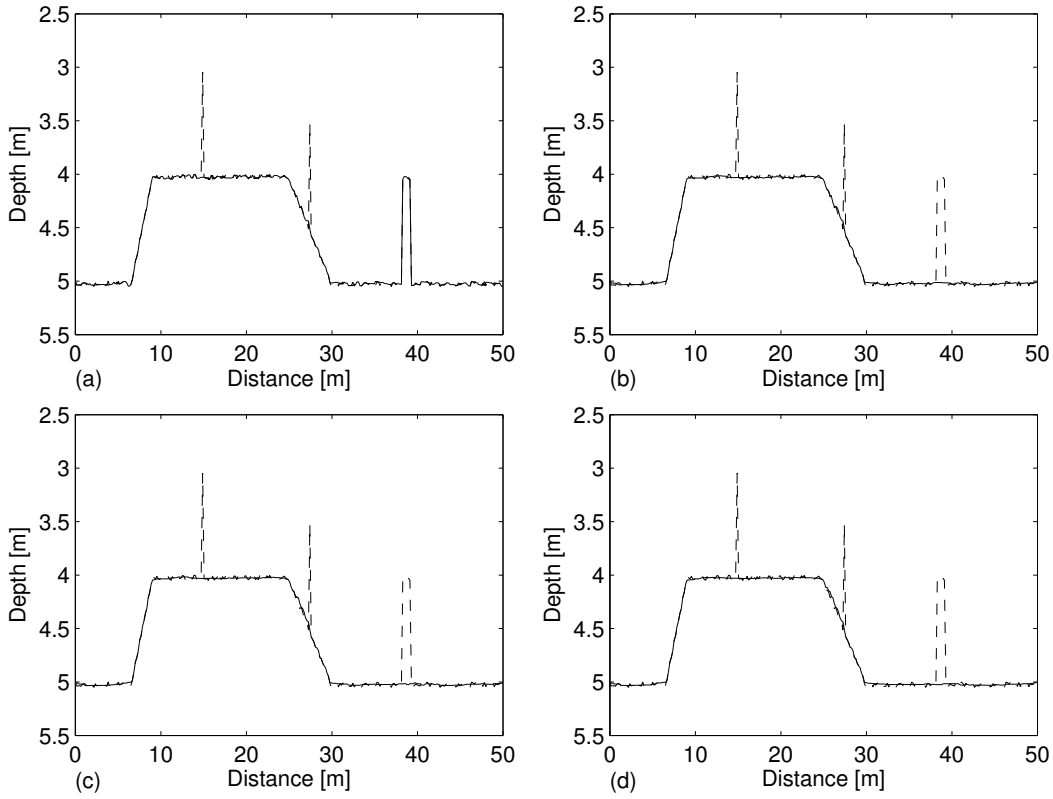

Figure 4.8: The dashed and solid lines represent the original and median-filtered signals, respectively. The following filter lengths have been used: (a) 3 ($N = 1$), (b) 13 ($N = 6$), (c) 17 ($N = 8$) and (d) 25 ($N = 12$).

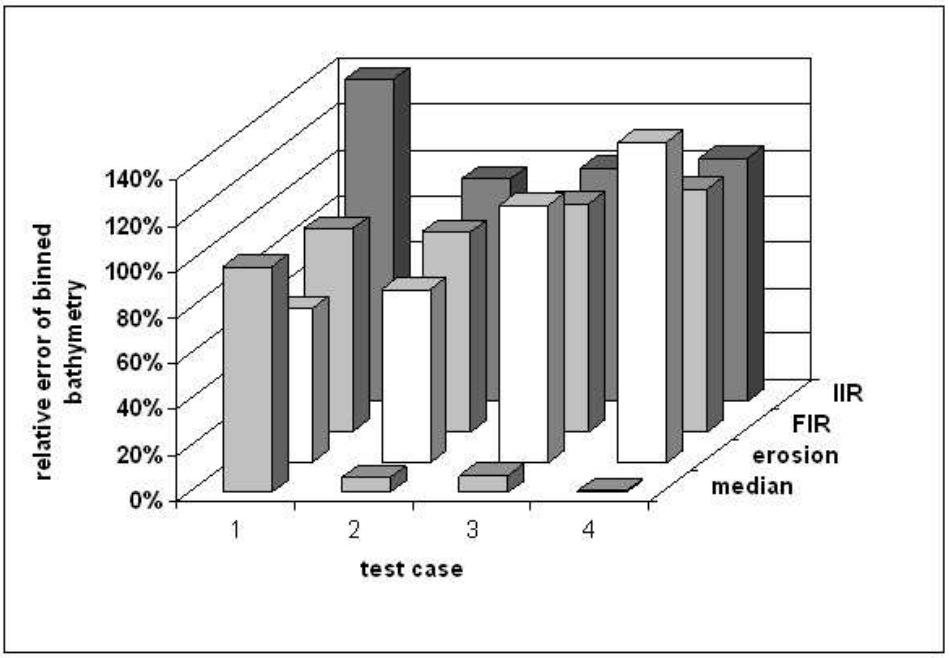

Figure 4.9: Illustration of the relative errors presented in Table 4.1. The numbers 1 through 4 for each method correspond to the order used in Table 4.1.

Filter type	Cutoff wave number [cm^{-1}]	Filter length	Relative error
FIR	7.5	-	88.3 %
FIR	15	-	87.0 %
FIR	30	-	99.2 %
FIR	45	-	105.8 %
IIR	7.5	-	139.9 %
IIR	15	-	97.0 %
IIR	30	-	101.6 %
IIR	45	-	105.6 %
Erosion	-	3 ($N = 1$)	66.9 %
Erosion	-	7 ($N = 3$)	75.2 %
Erosion	-	11 ($N = 5$)	118.1 %
Erosion	-	13 ($N = 6$)	139.4 %
Median	-	3 ($N = 1$)	98.5 %
Median	-	13 ($N = 6$)	6.6 %
Median	-	17 ($N = 8$)	7.7 %
Median	-	25 ($N = 12$)	9.0 %

Table 4.1: Relative errors between spike-free synthetic bathymetry and filtered bathymetries: comparison between filters.

Deciding the appropriate cutoff wave number will therefore depend upon both the characteristics of the data and the end use of the filtered bathymetry; it follows that linear filtering depends on the skill of the analyst and cannot be used in an entirely mechanistic manner.

While nonlinear filtering is clearly superior to linear filtering, practical application requires a systematic approach to choosing the appropriate filter order. For a signal with uniform data spacing it is possible to select the minimum nonlinear filter order based solely on the filter construction. To ensure that the median is not in a data spike, an effective median filter window must have more points outside the spike than are included in the spike. Thus, an order N filter (window size of order $2N + 1$) is guaranteed to remove an LWD spike

of N data points or fewer. An erosion filter requires only one point in the window that is outside the spike, so a filter of order N will remove spikes of $2N$ data points or fewer. For uniform data spacing of Δx and a maximum spike physical length scale of $\mathcal{L}$, it follows that the required median filter order is $N \geq \mathcal{L}/\Delta x$, and the required erosion filter order is $N \geq \mathcal{L}/(2\,\Delta x)$. For the synthetic bathymetry analyzed above, the largest data spike is precisely 0.96 m and the sample spacing is 0.16 m, so the minimum required median filter is $N = 6$ and the minimum required erosion filter is $N = 3$. These theoretical minimums match the results in Figures 4.7b and 4.8b showing the lowest order filters that remove all the data spikes. However, for a data set with non-uniform data spacing, the previous simple arguments for choosing the filter order cannot be applied unless the filter order is allowed to change as a function of the local data spacing. While this might be a possible approach, it was considered overly-complex for the present work. Instead, we investigated the use of a single filter order for an entire data set, where the filter order is selected by a statistical analysis of the data spacing and the expected scales of the LWD.

LWD in the Sulphur River (e.g., Figure 1.1 and Appendix B) is typically tree trunks and large limbs with diameters in the range of 10-20 cm, along with stumps and associated rootballs that may be more than a meter across. Allowing for the possible oblique angle of the survey boat with the LWD, we consider that a reasonable LWD length scale for filtering is $\mathcal{L} = 1$ m. This scale will also filter out similar scales of bathymetric roughness, but this is

unlikely to significantly affect the overall accuracy of the filtered bathymetry. In the Sulphur River survey, the mean data spacing was 16 cm, with a standard deviation of 7 cm. If the mean data spacing ($\overline{\Delta x}$) is used to select the minimum median filter order as $N \geq \mathcal{L}/(\overline{\Delta x})$, a minimum median filter order of $N = 6$ would result. Similarly, the minimum erosion filter order would be $N \geq \mathcal{L}/(2\,\overline{\Delta x})$, which results in $N = 3$. However, the possibility of slower boat speeds coinciding with the signal from a piece of LWD leads us to prefer a more conservative median filter order estimate based on

$$N_{\text{median}} \geq \frac{\mathcal{L}}{\overline{\Delta x} - \sigma} \tag{4.2}$$

where σ is the standard deviation of the survey data spacing. This provides a minimum median filter of $N = 11$ for the Sulphur River survey. For an erosion filter, the appropriate minimum is

$$N_{\text{erosion}} \geq \frac{\mathcal{L}}{2\left(\overline{\Delta x} - \sigma\right)} \tag{4.3}$$

resulting in an erosion filter of $N = 6$ for the Sulphur River survey. Filtered and raw data for three 100-meter sections of surveyed bathymetry using the minimum filters computed from equations 4.2 and 4.3 are shown in Figures 4.10 and 4.11. Additional experiments (not shown) were conducted to find the minimum median and erosion filter orders that removed all the data spikes. An $N = 11$ median filter and an $N = 6$ erosion filter are the minimum that visually remove all the large spikes in the data set, indicating that equations 4.2 and 4.3 are suitably conservative. Indeed, in the present case it is possible to argue

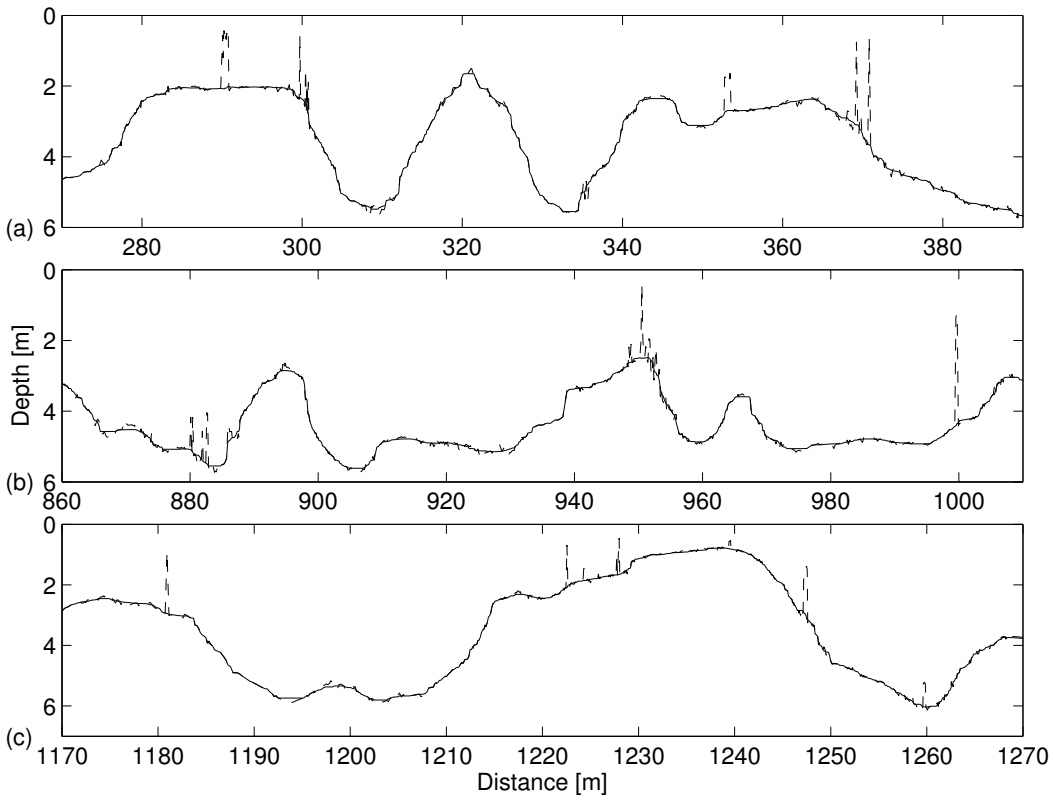

Figure 4.10: The dashed line and solid lines represent the original and filtered signals, respectively. A N=11 median filter has been used, as calculated by Eq. 4.2.

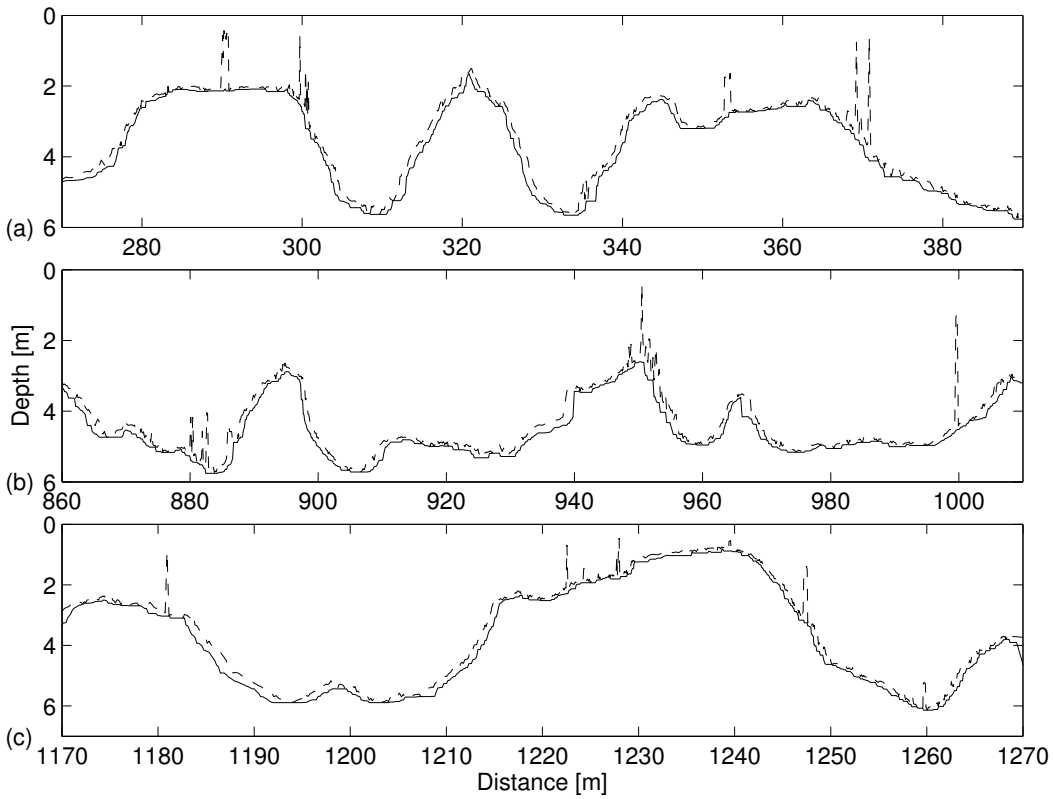

Figure 4.11: The dashed line and solid lines represent the original and filtered signals, respectively. A N=6 erosion filter has been used, as calculated by Eq. 4.3.

that the LWD length scale and the mean data spacing provides a good filter order – but this is unlikely to always hold true. Figures 4.10 and 4.11 show that both filters effectively remove all the LWD spikes, but the erosion filter produces unacceptably large erosion of the bathymetry throughout the signal. The performance difference between the two nonlinear filtering methods can be seen more clearly by plotting the filter effect – i.e. the absolute difference between the filtered signal and the raw bathymetry (Figure 4.12). Regions with steep bathymetry slopes without any sharp spikes (e.g., between 325 and 330 m), have continuous erosion of the bathymetry that peaks at 0.8 m for the erosion filter. In contrast, the effect of the median filter outside of the large, visually-identifiable spikes is more episodic and has a maximum of only 0.15 m. A comparison of three median filter orders has been performed and results are shown in Figure 4.13. While using $N = 6$ – obtained from the expression $N \geq \mathcal{L}/(\overline{\Delta x})$ – misses some broad spikes, a filter order of $N = 22$ – taken to be twice the order suggested by Eq. 4.2 – clearly erodes the original signal too severly. The order $N = 11$ computed by Eq. 4.2 is a successful compromise between those extreme situations.

From the foregoing, the erosion of steep slopes by an erosion filter would seem to preclude their use in filtering LWD. However, in a closely-packed debris field where the length scales of surveyed LWD are larger than distance between debris items, an erosion filter may be preferable. It has been shown that where impulses are separated by fewer than N samples, an order N nonlinear filter may only partially remove the impulses (Justusson, 1981). Thus, for a

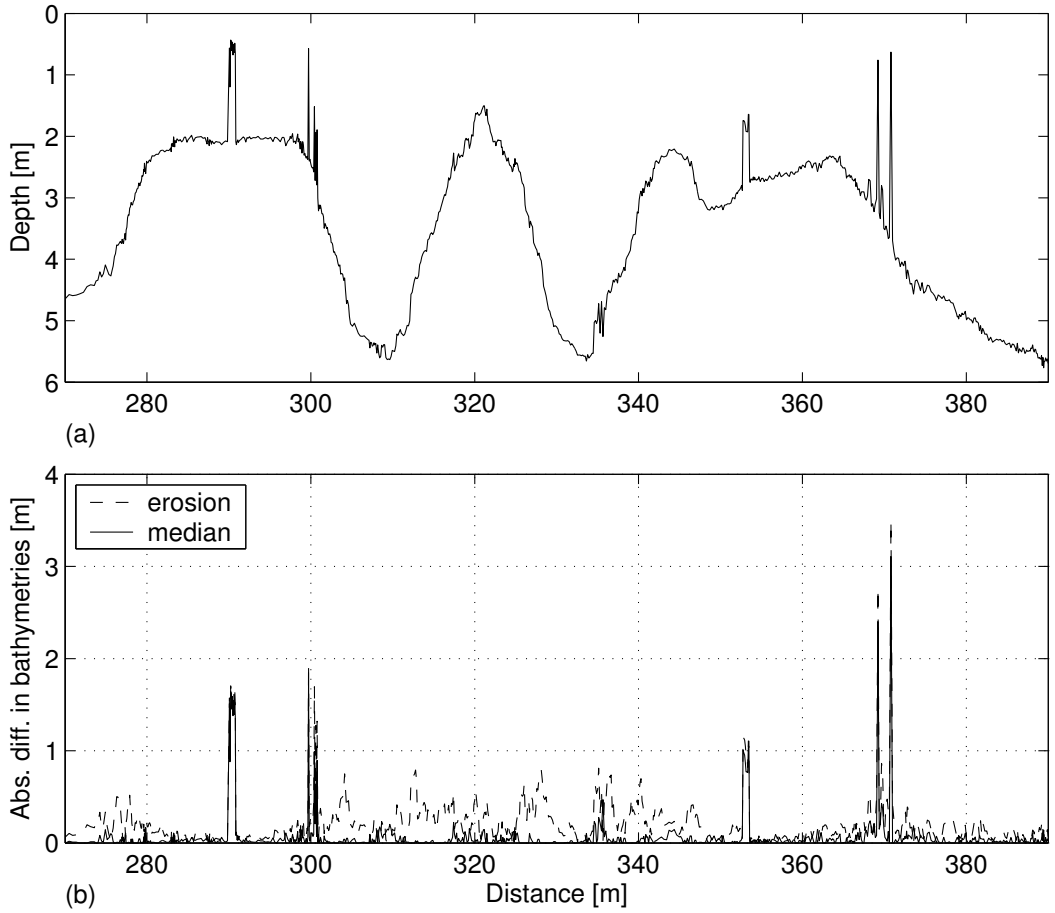

Figure 4.12: Evaluation of erosion ($N = 6$) and median ($N = 11$) filtering performance. (a) Original bathymetry. (b) Dashed and solid lines represent absolute differences between original and filtered bathymetries, for erosion and median filters, respectively.

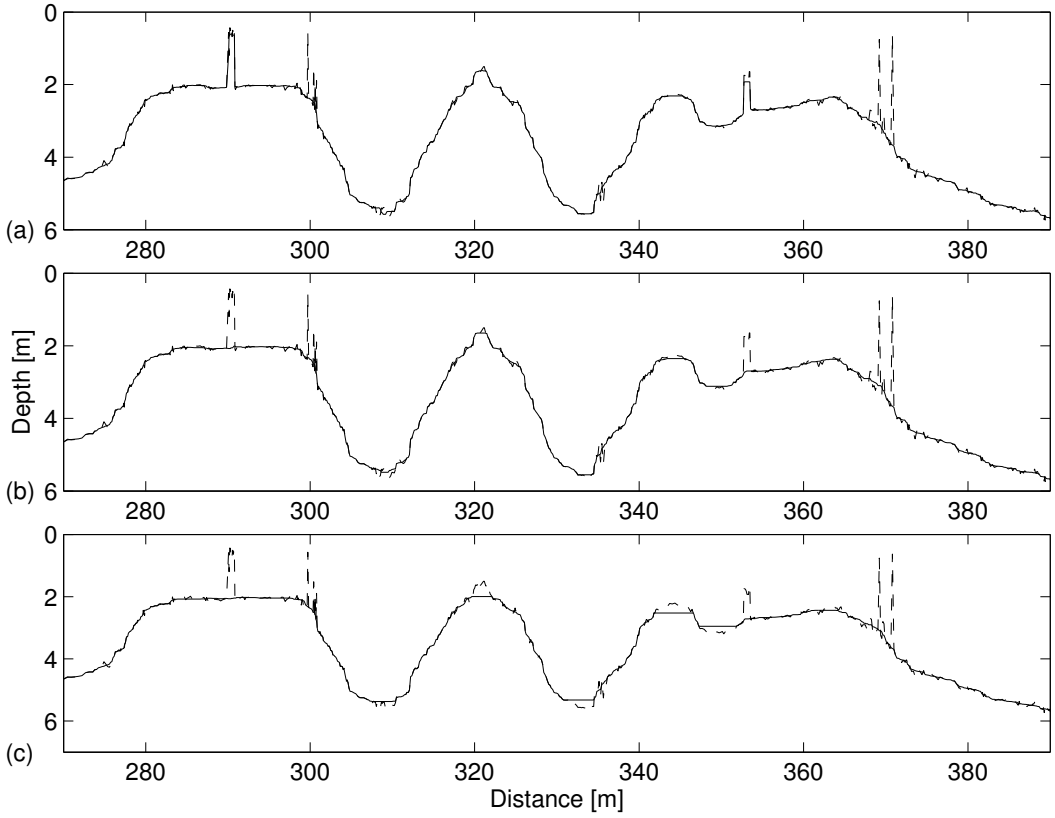

Figure 4.13: Comparison of three median filter orders. Frame (a) shows filtering with $N = 6$ obtained by using the expression $N \geq \mathcal{L}/(\overline{\Delta x})$. Frame (b) shows filtering with $N = 11$ obtained from Eq. 4.2. Frame (c) shows filtering with $N = 22$, taken to be twice the order used in the previous frame.

debris field with length scale $\mathcal{L}$, the median filter will require spacing between individual debris items greater than $\mathcal{L}$. In contrast, an erosion filter need be only half the order of a median filter for removing the same debris scales, so an erosion filter can be effectively employed where the debris with minimum separation of $\mathcal{L}/2$. Thus, for river reaches with sections of closely-packed debris, it may be effective to design a filtering algorithm that locally switches between median and erosion filter application, depending on the spacing of the data spikes. Developing an automated method for selective application of different filters remains an area requiring further research.

4.3 Conclusions

This chapter has examined the effectiveness of both linear and nonlinear filters for removing LWD signals from bathymetry data. Using a synthesized bathymetry, it was shown that linear filters require a trade-off between removing impulse signals and distorting naturally steep slopes in the background bathymetry. This trade-off makes linear filtering less desirable, and also inhibits development of an automated approach to removing LWD signals from bathymetry data (the selection of linear filter characteristics to remove LWD is a qualitative decision). It was demonstrated that nonlinear filtering, and specifically median filtering, is effective for removing LWD signals without distorting steep slopes. Nonlinear erosion filtering was shown to be less effective as the method's bias causes steep slopes to be eroded, with the scale of the erosion being a function of the filter order and the slope steepness. Nonlinear filtering was shown to be amenable to automatic selection of the filter order

using only statistics of the survey data spacing and the length scales of LWD. Nonlinear median filtering methods were shown to provide a practical means of removing the LWD signal from bathymetry data collected on the Sulphur River, Texas. As discussed in the previous chapter, the difference between the signal with LWD removed and the original bathymetry can be used to map LWD locations, a technique applied by Osting *et al* (2003) as part of an aquatic habitat analysis. Figure 4.14 shows the locations of LWD as determined by $N = 11$ median filtering. The entire bathymetric data set (shown in Figure 2.8) has been processed. Spikes exceeding 0.5 m in height have been considered LWD.

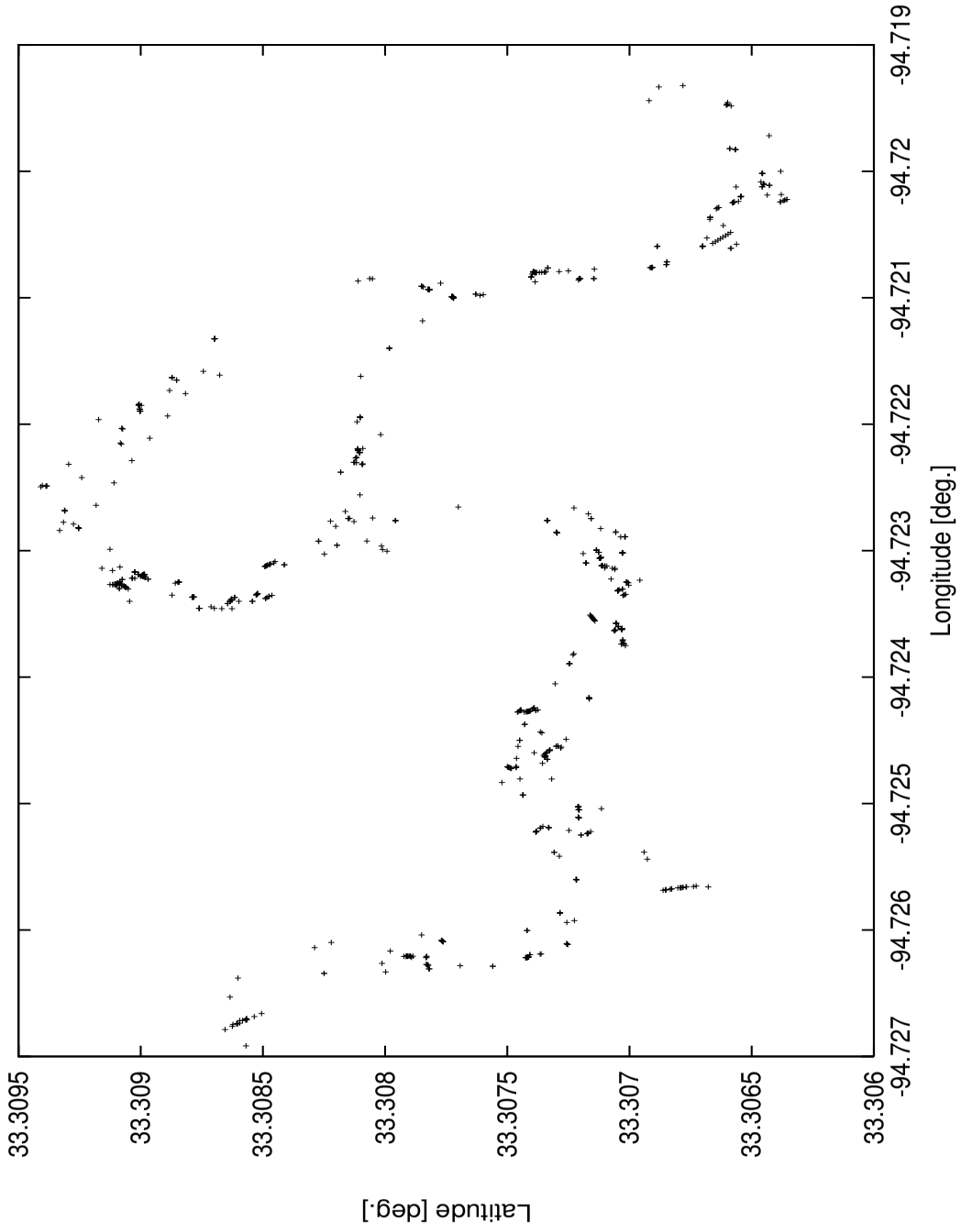

Figure 4.14: LWD locations for entire bathymetric data set as determined by $N = 11$ median filtering. Spikes exceeding 0.5 m in height have been considered LWD.

Chapter 5

General 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.

Among the former, the first method, σ-discrimination, was shown to be suitable for identifying likely LWD data points so that they can be separated from

the original data set. This provides a background bathymetry that is effectively free from LWD and is appropriate for modeling or GIS purposes. The principal drawback of this method is that selection of the discriminative factor F remains arbitrary and relies on visual inspection of the data. The second statistical method demonstrated, scale-space analysis, proved less successful in identifying LWD. Scale-space analysis for identifying LWD requires setting upper and lower windowing limits on the "fingerprint" width and smoothing height of "arches" associated with LWD.

We examined the effectiveness of both linear and nonlinear filters for removing LWD signals from bathymetry data. Using a synthesized bathymetry, it was shown that linear filters require a trade-off between removing impulse signals and distorting naturally steep slopes in the background bathymetry. This trade-off makes linear filtering less desirable. It was demonstrated that nonlinear filtering, and specifically median filtering, is effective for removing LWD signals without distorting steep slopes. Nonlinear erosion filtering was shown to be less effective as the method's bias causes steep slopes to be eroded. Nonlinear filtering was shown to be amenable to automatic selection of the filter order using only statistics of the survey data spacing and the length scales of LWD. Nonlinear median filtering methods were shown to provide a practical means of removing the LWD signal from bathymetry data collected on the Sulphur River, Texas. The difference between the signal with LWD removed and the original bathymetry can be used to map LWD locations, a technique applied by Osting *et al* (2003) as part of an aquatic habitat analysis.

Appendix A

Acronyms

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 B

Pictures of LWD in Sulphur River

This appendix contains more pictures of LWD in the Sulphur River. All pictures were taken during low flow.

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

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

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

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

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

Appendix C

Bathymetry Process 1.1: User's guide

C.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).

C.2 Installation

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

C.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.

C.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.

C.2.3 Completion

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

C.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*.

C.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).

C.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.

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.

[1]Refer to section C.4 for a brief overview of median filtering or section 4.1.2 for more detailed explanations.

C.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.

C.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 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).

C.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.

[2]Who wants to use Windows anyway ?

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

Appendix D

Bathymetry process: 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%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%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 %.10f\n",
```

```
                ksi, gps_positions[i].soundings[j], x, y );
          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 */
```

```c
/* ----------------------------------------------------------
 *
 *      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*/
```

```c
/* ----------------------------------------------------------
 *
 *      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 */
```

Appendix E

Scale-space filtering: code listing

```
/* ------------------------------------------------------------------------
 *
 *      main.c
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-10
 *
 *
 *      USAGE : [PREFIX] [N_LEVELS] [SIGMA]
 *       PREFIX   : prefix of the files to be read and written to
 *    N_LEVELS : number of levels of smoothing
 *    SIGMA    : standard deviation of Gaussian filter
 *
 * This program reads PREFIX.ori containing the bathymetry and
 *      applies a Gaussian filter of standard deviation SIGMA (N_LEVELS
 * times). It then browses successive levels of smoothened bathymetries
 * to identify troughs and peaks in order to produce a fingerprint
 *      (2-D image representing a plot of the displacement of troughs and
 *      peaks versus smoothing level), which can be read by MATLAB from
 *      the file PREFIX_fpr.m
 *
 * ------------------------------------------------------------------------ */

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

# include "defs.h"
# include "preprocess.h"
# include "process.h"
# include "postprocess.h"

int main ( int argc, char **argv )
{
   /* =========
    * Variables
    * ========= */

   double  KSI[MAX_BATH_SIZE];
   double  H[MAX_BATH_SIZE];
   double  X[MAX_BATH_SIZE];
   double  Y[MAX_BATH_SIZE];

   char    fname_in[MAX_FNAME_SIZE];
   char    fname_fpr[MAX_FNAME_SIZE];
   char    fname_flt[MAX_FNAME_SIZE];
   char    fname_lwd[MAX_FNAME_SIZE];

   int     i;
   int     j;
   time_t  tic, toc;
   double  **SM;     /* Smoothing matrix */
   int     **FEAT;      /* Matrix of features (peaks and troughs) */
   int   n_levels;
   double  sigma;
```

```
    int     N_bath;
    int     return_value;

    /* ======================================================
     * PREPROCESS  : Arguments, read bathymetry, init matrix
     * ====================================================== */

    tic = time ( NULL );

    /* Handle arguments */
    return_value =  handle_arguments ( argc, argv, fname_in,
          fname_fpr, fname_flt, fname_lwd, &n_levels, &sigma );
    if ( return_value == 0 )
       return 0;

    /* Read bathymetry */
    N_bath =  read_bathymetry( fname_in, KSI, H, X, Y);

    /* Initialize smoothing matrix */
    SM = ( double ** ) malloc ( (n_levels + 1) * sizeof ( double * ) );
    for ( i = 0 ; i < (n_levels + 1) ; i++ )
    {
       SM[i] = ( double * ) malloc ( N_bath * sizeof ( double ) );

       if ( SM[i] == NULL )
          return 0;
    }

    for ( i = 0 ; i < (n_levels + 1) ; i++ )
       for ( j = 0 ; j < N_bath ; j++ )
          SM[i][j] = 0.0;

    /* Initialize matrix of features */
    FEAT = ( int ** ) malloc ( (n_levels + 1) * sizeof ( int * ) );
    for ( i = 0 ; i < (n_levels + 1) ; i++ )
    {
       FEAT[i] = ( int * ) malloc ( N_bath * sizeof ( int ) );

       if ( FEAT[i] == NULL )
          return 0;
    }

    for ( i = 0 ; i < (n_levels + 1) ; i++ )
       for ( j = 0 ; j < N_bath ; j++ )
          FEAT[i][j] = 0;

    /* ======================================================
     * PROCESS     : Recursive application of Gaussian filter
     * ====================================================== */
    (void) gaussian_filter ( KSI, H, N_bath, SM, sigma, n_levels);

    toc = time ( NULL );

    /* ======================================
     * POSTPROCESS : Construction of fingerprint
     * ====================================== */
    (void) fingerprint ( KSI, N_bath, SM, FEAT, n_levels, fname_fpr, fname_flt );

    (void) identify_LWD ( KSI, H, X, Y, N_bath, FEAT, n_levels, fname_lwd );

    /* ===========
          * Free memory
     * =========== */

    for ( i = 0 ; i < (n_levels + 1) ; i++ )
       free ( SM[i] );
    free ( SM );

    for ( i = 0 ; i < (n_levels + 1) ; i++ )
       free ( FEAT[i] );
    free ( FEAT );

    printf ( "\nElapsed time : %ld s\n", toc - tic );

    return 1;
}
```

```
/* ----------------------------------
 *
 *      defs.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-13
 *      Last update      : 2003-06-13
 *
 * ---------------------------------- */

# define MAX_BATH_SIZE  65536
# define MAX_FNAME_SIZE 128
```

```
/* ----------------------------------
 *
 *      preprocess.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-13
 *
 * ---------------------------------- */

int handle_arguments ( int argc, char **argv,
             char *fname_in,
             char *fname_fpr,
             char *fname_flt,
             char *fname_lwd,
             int *n_levels, double *sigma );

int read_bathymetry ( char *prefix, double *KSI, double *H, double *X, double *Y );
```

```
/* ---------------------------------
 *
 *      preprocess.c
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-13
 *
 * --------------------------------- */

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

# include "preprocess.h"


int handle_arguments ( int argc, char **argv,
             char *fname_in,
             char *fname_fpr,
             char *fname_flt,
             char *fname_lwd,
             int *n_levels, double *sigma )
/*
   PRE  :
   POST :
*/
{
   if ( argc < 4 )
   {
      printf ( "Wrong number of arguments !\n" );
      printf ( "USAGE : [PREFIX] [N_LEVELS] [SIGMA].\n" );
      return 0;
   }

   /* Filenames */
   strcpy ( fname_in , argv[1] );
   strcpy ( fname_fpr, argv[1] );
   strcpy ( fname_flt, argv[1] );
   strcpy ( fname_lwd, argv[1] );


   strcat ( fname_in , ".ori"   );
   strcat ( fname_fpr, "_fpr.m" );
   strcat ( fname_flt, "_sl.m" );
   strcat ( fname_lwd, "_lwd" );

   /* Number of levels */
   *n_levels = (int) atoi ( argv[2] );

   /* Sigma */
   *sigma = (double) atof ( argv[3] );

   return 1;

} /* handle_arguments */


int read_bathymetry ( char *fname_in, double *KSI, double *H, double *X, double *Y )
/*
   PRE  :
   POST :
*/
{

   /* Variables */
   int   i;

   double  ksi;
   double  h;
   double  x;
   double  y;

   int     n_read;      /* Number of numbers read by fscanf */

   FILE  *fp_bath;

   printf ( "Reading bathymetry in file '%s'\n", fname_in );

   /* Open file */
        fp_bath = fopen ( fname_in, "r" );
```

```
        if (fp_bath == NULL)
        {
                printf ( "Opening file '%s' failed in 'read_bathymetry'.\n", fname_in );
                return 0;
        }

   /* Read bathymetry */
   i = 0;
   while ( !feof(fp_bath) )
   {
      n_read = fscanf ( fp_bath , "%lf %lf %lf %lf\n", &ksi, &h, &x, &y );

      KSI[i] = ksi;
      H[i]   = h;
      X[i]   = x;
      Y[i]   = y;

      i++;
   } /* End browsing fp_bath */

   fclose ( fp_bath );

   return i;

} /* read_bathymetry */
```

```
/* ----------------------------------
 *
 *      process.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-12
 *      Last update      : 2003-06-13
 *
 * ---------------------------------- */

int gaussian_filter ( double *KSI, double *H, int N, double **SM, double sigma, int n_levels );

double mean_vector ( double *V, int N );
```

```c
/* ----------------------------------
 *
 *      process.c
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-12
 *
 * ---------------------------------- */

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

# include "process.h"
# include "defs.h"

int gaussian_filter ( double *KSI, double *H, int N_bath, double **SM, double sigma, int n_levels )
/*
   PRE  :
   POST :
*/
{

   double delta;
   double mu;
   double diff1;
   double new_diff1;
   double diff2;
   double new_diff2;
   int    n1,n2;
   int    i;      /* Multi-purpose counter */
   int    j;      /* Multi-purpose counter */
   int    sl;     /* Smoothing level counter */
   double **EXT;     /* Filter information */
   double sum_G;
   int    k_left;
   int    k_right;
   double tmp1;      /* Factors to accelerate filtering calculation */
   double tmp2;
   double H_new[MAX_BATH_SIZE];  /* Bathymetry minus mean depth */


   /* ==================
        * Preparation of data
    * ================== */

   printf ( "Recursive gaussian filtering : \n" );

   delta = 4.0 * sigma;
   mu    = mean_vector ( H, N_bath );

   for ( i = 0 ; i < N_bath ; i++ )
      H_new[i] = H[i] - mu;

   /* ==========================================================
    * Find vector extremities between which filter is applicable
    * ========================================================== */
   printf ( "   --> Preparation of data\n" );


    /* First extremity */
    n1 = 0;
    diff1 = delta;

    new_diff1 = fabs ( KSI[n1+1] - KSI[0] - delta );
    while ( diff1 > new_diff1 )
    {
      diff1 = new_diff1;
      n1++;
      new_diff1 = fabs ( KSI[n1+1] - KSI[0] - delta );
    }

    if ( (KSI[n1] - KSI[0] - delta) < 0 )
      n1++;

   /* Second extremity */
    n2 = N_bath - 1;
    diff2 = delta;
```

```
 new_diff2 = fabs ( KSI[N_bath-1] - KSI[n2-1] - delta );
 while ( diff2 > new_diff2 )
 {
   diff2 = new_diff2;
   n2--;
   new_diff2 = fabs ( KSI[N_bath-1] - KSI[n2-1] - delta );
 }

 if ( (KSI[N_bath-1] - KSI[n2] - delta) < 0 )
   n2--;


/* ================================
 * Calculation of filter information
 * ================================ */
printf ( "   --> Calculation of filter information\n" );

/* Construction of matrix */
EXT = ( double ** ) malloc ( 3 * sizeof ( double * ) );
for ( i = 0 ; i < 3 ; i++ )
   EXT[i] = ( double * ) malloc ( (n2-n1+1) * sizeof ( double ) );

/* Initialization of filter information matrix  */
for ( i = 0 ; i < 3 ; i++ )
   for ( j = 0 ; j < (n2-n1+1) ; j++ )
      EXT[i][j] = 0.0;


tmp1 = 1 / (sigma*sqrt(2*M_PI));
tmp2 = 2*sigma*sigma;
for ( i = n1 ; i < n2+1 ; i++ )
{
   k_left  = 0;
   k_right = 0;

   while ( ( KSI[i] - KSI[i - k_left - 1] - delta ) < 0.0 )
      k_left++;

   while ( ( KSI[i + k_right + 1] - KSI[i] - delta ) < 0.0 )
      k_right++;

   sum_G = 0.0;
   for ( j = -k_left ; j < (k_right + 1) ; j++ )
      sum_G += exp( - ( KSI[i] - KSI[i+j] ) * ( KSI[i] - KSI[i+j] ) / tmp2 );

   sum_G *= tmp1;

   EXT[0][i - n1] = k_left;
   EXT[1][i - n1] = k_right;
   EXT[2][i - n1] = sum_G;
}


/* ========================================
      * Successive applications of gaussian filter
 * ======================================== */
printf ( "   --> Application of Gaussian filters \n" );
printf ( "       " );

/* Zeroeth level of the smoothing matrix is the original bathymetry */
for ( i = 0 ; i < N_bath ; i++ )
   SM[0][i] = H_new[i];

/* Recursive application of Gaussian filter */
for ( sl = 1 ; sl < (n_levels + 1) ; sl++ )
{
   for ( i = n1 ; i < n2 ; i++ )
   {
      k_left  = EXT[0][i - n1];
      k_right = EXT[1][i - n1];
      sum_G   = EXT[2][i - n1];

      /* Compute weighted average at position of index i*/
      for ( j = -k_left ; j < (k_right + 1) ; j++ )
      {
         SM[sl][i] += SM[sl-1][i+j] * exp( - ( KSI[i] - KSI[i+j] ) * ( KSI[i] - KSI[i+j] ) / tmp2 );
      }
      SM[sl][i] *= (tmp1 / sum_G);

   } /** End loop on positions **/
```

```
      printf ( "." );
      (void) fflush ( stdout );

   } /** End loop on smoothing levels **/
   printf ( "\n" );

   /* ============
         * Free memory
    * ============ */

   for ( i = 0 ; i < 3 ; i++ )
      free ( EXT[i] );
   free ( EXT );


   return 1;

} /* gaussian_filter */


double mean_vector ( double *V, int N )
/*

*/
{
   int   i;
   double   sum;

   sum = 0.0;
   for ( i = 0 ; i < N ; i++ )
      sum += V[i];

   return (sum / N);

} /* mean_vector */
```

```
/* ----------------------------------
 *
 *      postprocess.h
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-13
 *
 * ---------------------------------- */

char compare_sequence ( double h_left, double h_center, double h_right );

int fingerprint ( double *KSI, int N_bath,
        double **SM, int **FEAT,
        int n_levels,
        char *fname_fpr, char *fname_flt );

int identify_LWD ( double *KSI, double *H, double *X, double *Y, int N_bath,
         int **FEAT,
         int n_levels, char *fname_lwd );
```

```
/* ----------------------------------
 *
 *      postprocess.c
 *
 *      Laurent White
 *
 *      Date of creation : 2003-06-13
 *
 * ---------------------------------- */

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

# include "postprocess.h"


char compare_sequence ( double h_left, double h_center, double h_right )
/*
   PRE  : sequence of 3 numbers.
   POST : if the sequence is
            a positive slope : returns '+'
          a negative slope : returns '-'
          a trough         : returns 'T'
          a peak           : returns 'P'
          a zero slope     : returns '='
*/
{

   if ( ( h_left == h_center ) && ( h_center == h_right ) )
      return '=';

   if ( ( h_left <= h_center ) && ( h_center <= h_right ) )
      return '+';

   if ( ( h_left >= h_center ) && ( h_center >= h_right ) )
      return '-';

   if ( ( h_left > h_center ) && ( h_center < h_right ) )
      return 'T';

   if ( ( h_left < h_center ) && ( h_center > h_right ) )
      return 'P';

   return '0';

} /* compare_sequence */


int fingerprint ( double *KSI, int N_bath,
        double **SM, int **FEAT,
        int n_levels,
        char *fname_fpr, char *fname_flt )
/*
   PRE  :
   POST :
*/
{

   /* =========
    * Variables
    * ========= */

   FILE  *fp_fpr; /* File to export fingerprint */
   FILE    *fp_flt;  /* File to export smoothened signals */
   int   n_features; /* Count of features (trough/peak) within one smoothing level */
   int     i;
   int     j;
   char    c;


   /* =========
    * Open file
    * ========= */

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

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

   printf ( "Construction of fingerprint : \n" );

   /* ======================================================
    * Identify peaks and troughs within each smoothing level
    * ====================================================== */

   fprintf ( fp_fpr, "hold on\n" );
   fprintf ( fp_flt, "SL = [\n" );
   for ( i = 0 ; i < (n_levels + 1) ; i++ )
   {
      /* Treatement of a new smoothing level */
      n_features = 0;
      fprintf ( fp_fpr, "KSI%d = [", i );

      fprintf ( fp_flt, "%.8f ", SM[i][0] );
      /* Loop to read bathymetry data of current smoothing level */
      for ( j = 1 ; j < (N_bath - 1) ; j++ )
      {
         c = compare_sequence ( SM[i][j-1], SM[i][j], SM[i][j+1] );

         /* Append fingerprint file */
         if ( ( c == 'T' ) || ( c == 'P' ) )
         {
            fprintf ( fp_fpr, "%.8f ", KSI[j] );
            n_features++;
         }

         fprintf ( fp_flt, "%.8f ", SM[i][j] );

         /* Append matrix of features */
         if ( c == 'T' )
            FEAT[i][j] = 1;
         if ( c == 'P' )
            FEAT[i][j] = 2;

      } /* End loop treating current row */
      fprintf ( fp_flt, "%.8f ", SM[i][N_bath-1] );
      fprintf ( fp_flt, "\n" );

      fprintf ( fp_fpr, "];\n" );
      fprintf ( fp_fpr, "X%d=[", i );
      for ( j = 0 ; j < n_features ; j++ )
         fprintf ( fp_fpr, "%d ", i );

      fprintf ( fp_fpr, "];\n" );
      fprintf ( fp_fpr, "plot(KSI%d,X%d,'.','MarkerSize',\
         3,'MarkerFaceColor','b');\n", i, i );
      fprintf ( fp_fpr, "\n\n" );

      printf ( "." );
      (void) fflush ( stdout );

   } /* End loop treating rows */
   printf ( "\n" );
   fprintf ( fp_flt, "];" );

   /* Close stream */
   fclose ( fp_fpr );
   fclose ( fp_flt );

   /* Everything went well */
   return 1;

} /* fingerprint */


int identify_LWD ( double *KSI, double *H, double *X, double *Y,
         int N_bath, int **FEAT, int n_levels, char *fname_lwd )
/*
 * ARGS: KSI is the vector of curvilinear coordinates.
 *       SM is the matrix of smoothed levels.
 *       N_bath is the length of the bathymetry.
 *       FEAT is a matrix containing features associated
```

```
 *       with all smoothing levels (1 for a trough, 2 for a peak,
 *       0 otherwise). Note that the last smoothing level is the
 *       last line of the matrix.
 *       n_levels si the number of smoothing levels.
 *
 * RETURN: 1 if everything goes well, 0 if an error occurs.
 *
 * ACTION: Identification of arches in the fingerprint.
 *       This is perfomed as follows:
 *
 *
 * */
{
   int MAX_WIDTH;  /* Maximum width of arch summit */
   int MAX_DEV;    /* Maximum deviation in arch legs */
   int LOWEST_LEVEL; /* Lowest level in which search for arch summits */
   int i;      /* A counter */
   int j;      /* Another counter */
   int k;   /* Another one ... */
   int height; /* Arch height */
   double width;  /* Arch width */
   int j_next; /* Index of next peak or trough */
   int j_left; /* Index of left arch leg */
   int j_left_old; /* Index of left leg at previous level */
   int j_right;   /* Index of right arch leg */
   int j_right_old;/* Index of right leg at previous level */
   int f_left; /* Feature on the left of arch */
   int f_right;   /* Feature on the right of arch */
   int arch;   /* Indicates if the beginning of an arch is found */
   int index_lwd;  /* Index of LWD location */
   FILE *fp_lwd;   /* File for LWD locations */

   /* Arch is considered caused by LWD if the geometry
    * falls within the following limits. */
   double min_width;
   double max_width;
   int min_height;
   int max_height;

   fp_lwd = fopen ( fname_lwd, "w" );
   if ( fp_lwd == NULL )
   {
      printf ( "Error while opening file '%s'. Aborting.", fname_lwd );
      return 0;
   }

   printf ( "\nARCHES IDENTIFICATION\n" );
   printf ( "---------------------\n\n" );

   /* Ask the user to enter parameters */
   /* Maximum summit width: */
   printf ( "Enter maximum arch summit width (in terms of samples): " );
   scanf ( "%d", &MAX_WIDTH );

   /* Maximum deviation in arch legs: */
   printf ( "Enter maximum deviation in arch legs (in terms of samples): " );
   scanf ( "%d", &MAX_DEV );

   /* Lowest level in which search for arch summit must be perfomed: */
   printf ( "Lowest level for search of arch summit: " );
   scanf ( "%d", &LOWEST_LEVEL );

   /* Get LWD geometric specifications */
   printf ( "Min width [cm]: " );
   scanf ( "%lf", &min_width );

   printf ( "Max width [cm]: " );
   scanf ( "%lf", &max_width );

   printf ( "Min height [in terms of smoothing levels]: " );
   scanf ( "%d", &min_height );

   printf ( "Max height [in terms of smoothing levels]: " );
   scanf ( "%d", &max_height );


   /* Loop through smoothing levels (from upper down !) */
   for ( i = n_levels ; i > LOWEST_LEVEL-1 ; i-- )
   {
      printf ("Smoothing level %d\n",i);
```

```
/* Loop to read features of current smoothing level */
j = 1;
while ( j < (N_bath - 1) )
/*for ( j = 1 ; j < (N_bath - 1) ; j++ )*/
{
   height = 0;
   width = 0;

   arch = 0;
   if ( (FEAT[i][j] == 1) || (FEAT[i][j] == 2) )
   /* We've got a peak or a trough.
    * Now, search the next one. */
   {
      j_next = j;
      while ( j_next < (N_bath-1) )
      {
         j_next++;
         if (FEAT[i][j_next] + FEAT[i][j] == 3)
         {
            if ( j_next - j + 1 <= MAX_WIDTH )
               arch = 1;
            break; /* Got a potential arch */                    }
      }

      /* At this point, if arch is 1, we have
       * a potential beginning of arch. Otherwise,
       * there is no valid arch. If an arch has
       * been found, we may continue the loop
       * after the second leg of the arch. */
   }

   if ( ( arch == 1 ) && ( i < n_levels ) )
   {

      /* Check if features really constitute
       * beginning of arch - that is, check if
       * other features exist within the above level
       * above the pair of features that have been
       * found */

      /* Search the level above the current one (i) */
      k = i + 1;

      /* LEFT LEG */

      /* Look on the left */
      j_left = j;
      while ( FEAT[k][j_left] != f_left )
      {
         j_left--;
         if ( j_left == 0 )
            break;
      }

      if ( j - j_left + 1 > MAX_DEV )
      /* Unable to find left leg on the left.
       * Now, we have to look on the right. */
      {
         j_left = j;
         while ( FEAT[k][j_left] != f_left )
         {
            j_left++;
            if ( j_left == N_bath - 1 )
               break;
         }
         if ( j_left - j + 1 > MAX_DEV )
         /* Lost track */
            j_left = -1;
      }

      /* RIGHT LEG */

      /* Look on the left */
      j_right = j_next;
      while ( FEAT[k][j_right] != f_right )
      {
         j_right--;
         if ( j_right == 0 )
            break;
      }
```

```
   if ( j_next - j_right + 1 > MAX_DEV )
   /* Unable to find right leg on the left.
    * Now, we have to look on the right. */
   {
      j_right = j_next;
      while ( FEAT[k][j_right] != f_right )
      {
         j_right++;
         if ( j_right == N_bath-1 )
            break;
      }
      if ( j_right - j_next + 1 > MAX_DEV )
      /* Lost track */
         j_right = -1;
   }

   if ( ( j_left != -1 ) || (  j_right != -1 ) )
   /* Then, the pair extends above the current pair
    * of found features ==> this is not the beginning
    * of an arch. */
      arch = 0;

   /* === END CHECKING WHETHER THE PAIR OF FEATURES
    * REALLY CONSTITUTES THE BEGINNING OF ARCH === */
}

/* Continue if valid arch was found */
if ( arch == 1 )
{

   /* At this point, we've got a valid
    * beginning of arch */
   f_left = FEAT[i][j];
   f_right = FEAT[i][j_next];
   j_left_old  = j;
   j_right_old = j_next;

   printf ( "Arch located between %.8f and %.8f. ",
       KSI[j], KSI[j_next] );
   printf ( "Tracking it down...\n" );


   /* Track arch down from current level to level 1 */
   for ( k = i-1 ; k > 0 ; k-- )
   {

      if ( (j_left_old == 0)
           || (j_right_old == N_bath-1))
      {
         printf("AAAAAAAA\n");
         (void) fflush (stdout);
         break;
      }

      /* LEFT LEG */

      /* Look on the left */
      j_left = j_left_old;
      while ( FEAT[k][j_left] != f_left )
      {
         j_left--;
         if ( j_left == 0 )
            break;
      }

      if ( j_left_old-j_left+1 > MAX_DEV )
      /* Unable to find left leg on the left.
       * Now, we have to look on the right. */
      {
         j_left = j_left_old;
         while ( FEAT[k][j_left] != f_left )
         {
            j_left++;
            if ( j_left == N_bath-1 )
               break;
         }
         if ( j_left-j_left_old+1 > MAX_DEV )
         /* Lost track */
            j_left = -1;
      }
      /* Update index of left leg at 'previous'
```

```
    * level.*/
   j_left_old = j_left;


   /* RIGHT LEG */

   /* Look on the left */
   j_right = j_right_old;
   while ( FEAT[k][j_right] != f_right )
   {
      j_right--;
      if ( j_right == 0 )
         break;
   }

   if ( j_right_old-j_right+1 > MAX_DEV )
   /* Unable to find right leg on the left.
    * Now, we have to look on the right. */
   {
      j_right = j_right_old;
      while ( FEAT[k][j_right] != f_right )
      {
         j_right++;
         if ( j_right == N_bath-1 )
            break;
      }
      if ( j_right-j_right_old+1 > MAX_DEV )
      /* Lost track */
         j_right = -1;
   }
   j_right_old = j_right;


   /* Increment arch height */
   if ( j_left == -1 || j_right == -1 )
   {
      /* Not a valid arch */
       height = 0;
       break;
   }

   height++;

} /* End tracking down arch */


if ( height > 0 )
{
   /* Width in [cm] */
   width = (KSI[j_right] - KSI[j_left]) * 100;

   printf ( "--> Arch located between %.8f and %.8f (height = %d, width = %.1f cm)\n",
    KSI[j_left], KSI[j_right],
    height, width );

}

if ( height > 0 )
/* Check if arch is caused by LWD */
{
   if ( ( width >= min_width   ) &&
        ( width <= max_width   ) &&
        ( height >= min_height ) &&
        ( height <= max_height ) )
   {

      /* Index of LWD location:
       * we take the index of
       * the trough.*/
      if ( f_left == 1 )
         index_lwd = j_left;
      else
         index_lwd = j_right;

      fprintf ( fp_lwd, "%.8f %.8f %.8f %.8f\n",
           KSI[index_lwd],
           H[index_lwd],
           X[index_lwd],
           Y[index_lwd]);
   }
```

```
            }

            j = j_next;
          } /* End treating valid arch */

          j++;

        } /* End loop treating current row */

    } /* End loop treating rows */

    fclose ( fp_lwd );
    return 1;

} /* identify_LWD */
```

Bibliography

Abbe, T. B., Montgomery, D. R. 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.

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

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

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

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

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

Dougherty, E. R. and Astola, J. T. 1999. Nonlinear filters for image processing. *SPIE/IEEE Series on Imaging Science & Engineering.*

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

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

Gerhard, M., Reich, M. 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.

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

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

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

Harmon, M. E., Franklin, J. F., Swanson, F. J., Gregory, S. V., Lattin, J. D., Anderson, N. H., Cline, S. P., Aumen, N. G., Sedell, J. R., Lienkaemper, G. W., Cromack Jr, K. and Cummins, K. W. 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.

Hodges, B. R. 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.

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

Jain, S. C. and Kennedy, J. F., 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..*

Justusson, B. I. 1981. Median filtering: statistical properties. *Topics in Applied Physics. Two-Dimensional Digital Signal Processing II: Transforms and Median Filters. Springer-Verlag.*

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

Kemp, J. L., Harper, D. M., Crosa, G. A. 2000. The habitat-scale ecohydraulics of rivers. *Ecological Engineering.* Vol. **16**, 17-29.

King, R., Ahmadi, M., Gorgui-Naguib, R., Kwabwe, A. and Azimi-Sadjadi, M. 1989. Digital filtering in one and two dimensions: design and applications. *Plenum Publishing Corporation.*

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

Lisle, T. E. 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.

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

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

Minshall, G. W. 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.

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

Mutz, M. 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.

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

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

Oliver, M. A. and Webster, R. 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).*

Osting, T. 2003. An improved anisotropic scheme for interpolating scattered bathymetric data points in sinuous rivers channels. (TWDB Draft submitted to Dr. Paul F. Hudson, GRG384C Watershed Systems and Environmental Management) *Texas Water Development Board.*

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

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

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

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

Sullivan, K., Lisle, T. E., Dolloff, C. A., Grant, G. E. and Reid, L. 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.

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

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

Vita

Laurent White was born in Uccle, Belgium on 10 November, 1979, the son of Annie Frère and Allen White. After completing his work at Collège Cardinal Mercier (Braine L'alleud, Belgium) in 1997, he entered the Université Catholique de Louvain in Louvain-La-Neuve, Belgium. He received a Diploma in Engineering in Applied Mathematics in June, 2002. In September 2002, he entered The Graduate School at the University of Texas.

Permanent Address: 13, Rue du Cortil Bailly
1380 Lasne
Belgium

This thesis was typeset in LaTeX by the author.