A sparse, bound-respecting parametrization of velocity models

Eric Dussaud and William W. Symes

ABSTRACT

We present a parsimonious representation of velocity models which allows for user-defined placement of nodes. Mild restrictions are imposed on the data structure so that a computationally efficient algorithm can be used to smoothly approximate nodal values on finely sampled regular grids. The building block of the algorithm operates on one-dimensional arrays, allows for user-defined control of smoothness and guarantees that bounds are preserved. The specific data structure allows to carry out this process recursively to obtain multi-dimension smooth and regularly gridded velocity models.

INTRODUCTION

A central problem of seismic modeling, imaging, and inversion techniques based on the Born approximation is that of finding an accurate background velocity model. The choice of model space to describe the background medium is therefore essential. On one hand the parametrization should be sparse, to specify detail only where necessary and avoid oversampling where the velocity varies only slightly. Unstructured meshes where the velocity is specified at nodal values have typically been used for that purpose. On the other hand, many seismic processing techniques (e.g. finite difference schemes) require that the velocity field be represented on finely sampled regular grids and satisfy some smoothness property. In particular, the background should be twice continuously differentiable to justify the use of high-frequency asymptotics [Beylkin, 1985; Rakesh, 1988] inherent to many seismic processing techniques. This is typically achieved in practice by cubic spline interpolation [Brandsberg-Dahl et al., 2003]. The major drawback of cubic spline interpolation is that explicit bounds which may be imposed on the nodal values are not guaranteed to be satisfied by the respective interpolants. Such explicit bounds are, for instance, essential for stability issues in finite difference schemes, or for controlling stretch and aliasing in diffraction sum implementations.

In this paper, we propose a very flexible way of describing the medium characteristics, by means of a parsimonious parametrization of the velocity field. Although not completely

unstructured, it does provide considerable freedom to allocate nodes to zones of rapid change in the model. Moreover, it is specifically tailored for a computationally efficient algorithm to smoothly approximate nodal velocity values on regular grids, for models of arbitrary dimensions, thereby preserving the order of explicit bounds on the velocity. We show that the above requirements can be met by first interpolating the irregularly sampled data on a sufficiently fine regular grid, and then applying a convolution operator with an appropriate kernel to smooth the interpolated values.

A SPARSE VELOCITY MODEL

The originality of the model representation lies in the choice of the particular parametrization chosen. In 3-D, the continuous velocity field $v(x, y, z)$ is defined relative to an orthogonal coordinate system with x and y as the horizontal axes and z as the vertical axis. By analogy with the typical marine seismic acquisition geometry, the x-axis is referred to as the in-line axis, while the y-axis is referred to as the cross-line axis. The velocity field is sampled at coordinate triplets (y_i, x_{ij}, z_{ijk}), i.e. $v_{ijk} \equiv v(y_i, x_{ij}, z_{ijk})$.

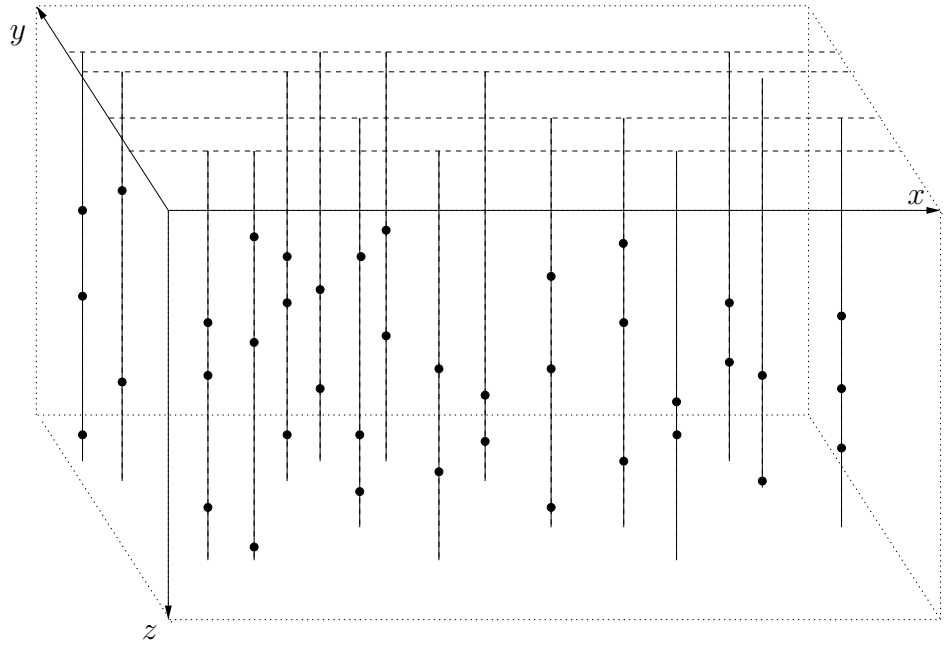

FIG. 1. Partially Irregular Grids (`PIGrid`s).

Figure 1 displays an example of such a velocity model. Each node on the diagram corresponds to a sample of the velocity field at a location (y_i, x_{ij}, z_{ijk}). This hierarchical data structure can be described as follows. To each cross-line coordinate y_i corresponds a x-z plane which in turns contains several "vertical wells", one for each in-line coordinate x_{ij}. Each of these wells contains many samples of the velocity at depth coordinates

z_{ijk}. Because the resulting grids are non entirely unstructured, we dubbed them *Partially Irregular Grids* (`PIGrid`s).

Although the parametrization imposes a certain level of structure, it does allow for **user-defined** placement of the nodes and thus allows for the sparse representation of complex geologic structures. Variations in depth can be represented arbitrarily well, just as lateral variations along the in-line direction can. Some limitations are imposed on how well lateral variations along the cross-line direction are modeled, but this is not a severe limitation in practice.

As we explained in the introduction, this sparse model representation cannot be used *as is* by most seismic applications. Rather, it is usually necessary that the nodal values be smoothly interpolated onto a regular grid. The procedure used for the smooth interpolation should also guarantee that explicit bounds imposed on the sparse model parametrization carry on to the interpolated model. The next section describes a simple 1-D scheme to do so.

1-D SMOOTH APPROXIMATION

In this section, we consider the problem of interpolating and smoothing irregularly spaced nodal values onto a 1-D regular grid. An example is shown on Figure 2. Again, each node on the diagram corresponds to a sample of the velocity field at that location. An example of an output regular grid is also shown on the diagram.

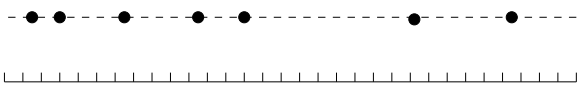

FIG. 2. The 1-D interpolation problem

The simplest piecewise polynomial interpolation is piecewise linear interpolation where linear polynomials are used on each interval. This just means that the nodal points are joined by straight lines. Formally, suppose that the regular grid consists of $N+1$ equally spaced points $z_0, \ldots, z_N$ that satisfy $z_0 < z_1 < \ldots < z_N$. Similarly, suppose that each node on Figure 2 represents a pair $(\bar{z}_k, \bar{v}_k)$, where $\bar{v}_k \equiv \bar{v}(z_k)$ is a sample of the field at location z_k. We further assume that there are $M+1$ such nodes ordered as $\bar{z}_0 < \bar{z}_1 < \ldots < \bar{z}_M$. Then the values of the velocity field on the regular grid are obtained as follows:

$$v(z_i) = \left[1 - \frac{z_i - \bar{z}_k}{\bar{z}_{k+1} - \bar{z}_k}\right] \bar{v}_k + \frac{z_i - \bar{z}_k}{\bar{z}_{k+1} - \bar{z}_k} \bar{v}_{k+1}, \quad \bar{z}_k \le z_i \le \bar{z}_{k+1} \tag{1}$$

Values at grid points outside the range $[\bar{z}_0, \bar{z}_M]$ are simply obtained by constant extension:

$$v(z_i) = \bar{v}_0 \quad \text{for} \quad z_i < \bar{z}_0 \quad \text{and} \quad v(z_i) = \bar{v}_M \quad \text{for} \quad z_i > \bar{z}_M \tag{2}$$

Before describing our method for smoothing piecewise linearly interpolated data, we briefly review how a convolution operator can be used to make a function smoother. In particular, we show how to construct B-splines using successive convolution of the box function [Trefethen, 1996]. The convolution of two functions functions u and v is the function $u * v$ defined by:

$$(u * v)(x) \equiv \int_{-\infty}^{\infty} dy\ u(x-y)v(y) = \int_{-\infty}^{\infty} dy\ u(y)v(x-y), \tag{3}$$

assuming these integrals exist. Note that the convolution operation amounts to a moving average of values $u(x)$ with weights defined by $v(x)$, or vice-versa. Suppose now that u is the function

$$u(x) = \begin{cases} \frac{1}{2} & \text{for} \quad -1 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

From the definition (3), it can be verified that

$$(u * u)(x) = \begin{cases} \frac{1}{2}(1 - |x|/2) & \text{for} \quad -2 \le x \le 2 \\ 0 & \text{otherwise} \end{cases}$$

and that

$$(u * u * u)(x) = \begin{cases} \frac{3}{8} - \frac{1}{8}x^2 & \text{for} \quad -1 \le x \le 1 \\ \frac{1}{16}\left(9 - 6|x| + x^2\right) & \text{for} \quad 1 \le |x| \le 3 \\ 0 & \text{otherwise} \end{cases}$$

The three functions u, $u * u$ and $u * u * u$ are plotted on Figure 3. Clearly, any function

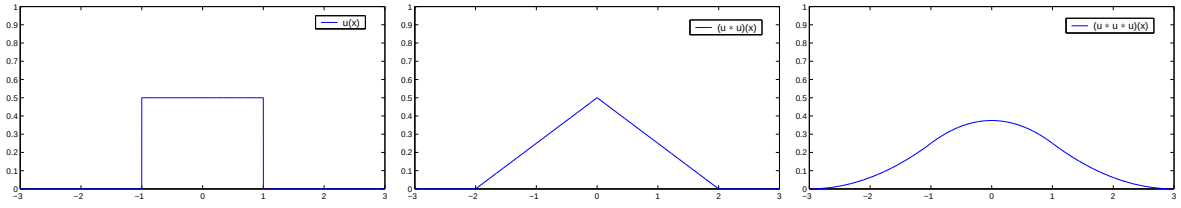

FIG. 3. B-spline construction

convolved with the function u becomes smoother, since the convolution amounts to a local moving average. In the above example, u is piecewise continuous, and it is easily seen that $u * u$ is continuous and has a piecewise continuous first derivative, and that $u * u * u$ has a continuous derivative and a piecewise continuous second derivative. In fact, a convolution $u_{(p)}$ of p copies of u is a piecewise polynomial of degree $p-1$ with a continuous $(p-2)$nd derivative and a piecewise continuous $(p-1)$st derivative, and is known as a B-spline [Trefethen, 1996].

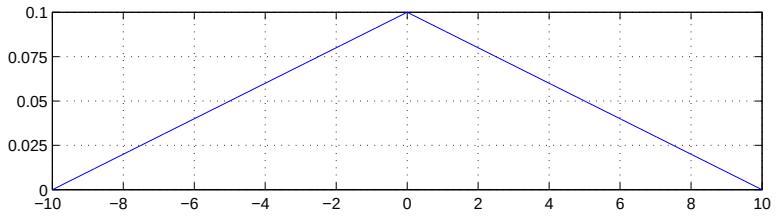

FIG. 4. Triangular kernel of width $h = 20$

Going back to our original problem, the above example shows that the convolution of any function with the triangular kernel of Figure 3 (central panel) yields a cubic spline. Therefore, to obtain a smooth (twice continuously differentiable) 1-D velocity model from the piecewise linear function defined by (1), we can perform a 1-D convolution with a triangular kernel similar to the function $u * u$ described above. The general form of the kernel can be written as:

$$k(x) = \begin{cases} \dfrac{2}{h}\left(1 - \dfrac{|x|}{h/2}\right) & \text{for} \quad -\frac{h}{2} \leq x \leq \frac{h}{2} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The scalar h is a **user-defined** parameter which represents the smoothing width of the convolution kernel: the larger h, the wider the kernel, and the more local averaging is performed during the convolution. The ratio $2/h$ appearing in (4) is a normalization factor determined so that:

$$\int dx\ k(x) = \int_{-h/2}^{h/2} dx\ \left[\frac{2}{h}\left(1 - \frac{|x|}{h/2}\right)\right] = 1.$$

In the example of Figure 3 (central panel), we had $h = 4$. An example of a kernel of width $h = 20$ is shown on Figure 4.

Given the values of the velocity field obtained using (1) on a grid $z_0 < z_1 < \ldots < z_N$ with sampling interval Δz, the smoothed velocity field is obtained by extending the grid on both sides by an amount equal to the smoothing width h, and then performing the convolution with kernel given by (4) on this extended grid. The discrete version of (3) takes the form:

$$v_s\left[q\Delta z\right] \approx \Delta z \sum_{p=-P}^{P+N} v\left[p\Delta z\right] k\left[(p-q)\Delta z\right], \quad q = 0, \ldots, N$$

Here $P = h/\Delta z$ represents the number of discrete points that are added on each side of the 1-D regular grid in order to perform the convolution, and v_s denotes the smoothed interpolated velocity field. An example of this 1-D smooth interpolation scheme is shown on Figure 5. The sparse velocity model consists in this case of four nodal values. The

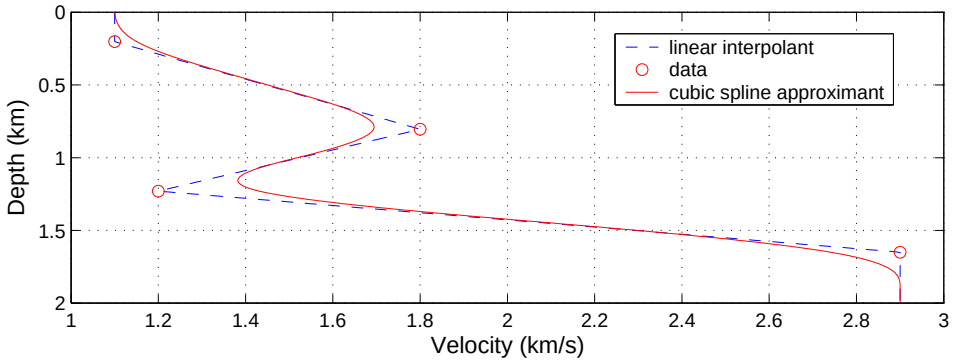

FIG. 5. Piecewise linear interpolation followed by smoothing

output regular grid consists of 501 points, 4m apart with $z_0 = 0$m, and $z_N = 2$km. The smoothing width of the convolution operator is $h = 500$m.

A key property of the scheme described above is that the resulting cubic spline is **not** an interpolant of the original data, but rather an approximant (see Figure 5). This is in contrast to standard cubic spline interpolation. The advantage of this construction stems from the fact that it guarantees that the order is preserved, in the sense that the approximant computed using bounds on nodal values is itself a bound for the approximant obtained using the original data. Figure 6 provides a simple illustration of this basic fact.

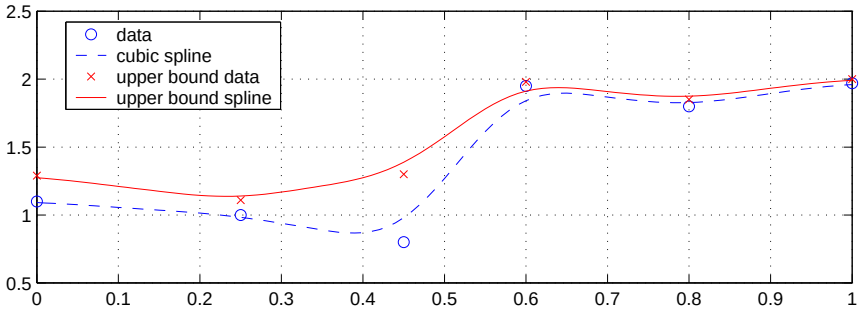

FIG. 6. Explicit bounds for the cubic spline approximant

The model consists of six nodal values (blue circles). The cubic spline approximant is shown as the dotted blue line. We then define an upper bound for the nodal values, that is, we keep the same location for the nodes but increase the associated values (red crosses in Figure 6). The corresponding cubic spline approximant is shown as the red solid line. Note that the order is preserved, i.e. the red line is an (uniform) upper bound for the blue line. Using the same exact data, the cubic spline interpolants (thereby using standard cubic spline interpolation) are displayed on Figure 7. In this case, the cubic

spline interpolating the red crosses is not an uniform upper bound for the cubic spline interpolating the original data. The order is not preserved in this case.

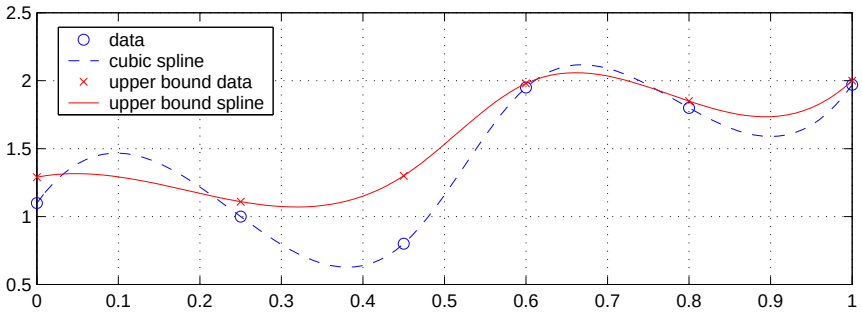

FIG. 7. Explicit bounds for the cubic spline interpolant

The fact is that the scenario depicted in Figure 6 is guaranteed by the method. The upper bound approximant is not *a priori* an upper bound for the original data (see the node located at $x = 0.6$ in Figure 6). However, it is guaranteed to be an upper bound for the cubic spline approximant computed with the original data. This is because the convolution operator performs local averaging of piecewise linear functions.

EXTENSION TO MULTIDIMENSION BY RECURSION

We describe in this section the recursive algorithm used to perform the smooth approximation of 2-D and 3-D velocity models. In the 3-D case, the output (regular) grid is described by the total number of desired samples n_z, n_x and n_y in the depth, in-line and cross-line directions, respectively, along with the associated sampling intervals Δz, Δx, and Δy. The core of the algorithm consists precisely of the scheme that we described in the previous section, namely piecewise linear interpolation followed by smoothing (via convolution) of 1-D arrays. We will subsequently refer to this scheme as the `SMPL` operator. The rest of the algorithm consists of re-arranging the data into 1-D irregular samples which can then be treated by the `SMPL` operator. Considering the `PIGrid` example displayed on Figure 1, the first step of the algorithm is rather simple: apply the `SMPL` operator to each "vertical well" corresponding to a point (y_i, x_{ij}) on the surface. The result is shown on Figure 8. Note that each vertical well has the same structure, namely it holds the same number of samples with an identical sampling interval Δz. The second step in the algorithm is illustrated on Figure 9, which displays a typical in-line section of the model shown on Figure 8. This step consists of looping through the discrete z-axis, forming irregularly samples at each level, and interpolating onto the regular samples along the x-axis. Thus, there are a total of n_z `SMPL` operations to perform at this stage. It is important to note that the nodes (there are 4 of them in the example of Figure 9) remain the same throughout this second step. Only the values of the velocity field at the nodes

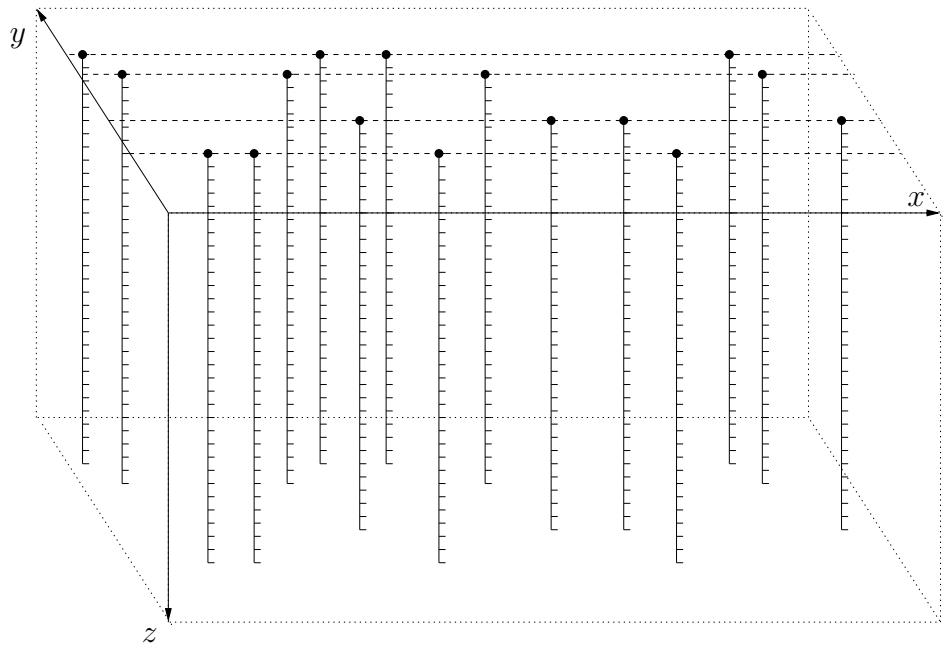

FIG. 8. Model after one step of `SMPL`

change from one level to the next. The resulting model is shown on Figure 10 (we only display two planes for clarity). The third step in the algorithm proceeds in exactly the same way. The iteration in this case is performed over the 2-D regular grid specified in the x-z plane, and the irregular samples are formed along the cross-line direction. Note that this step requires $n_x \cdot n_z$ applications of the `SMPL` operator. The resulting model consists of 3-D regularly gridded and smooth velocity data.

The data structure used to implement `PIGrid`s is described in more detail in the Appendix. It has been designed so that the algorithm carrying out the procedure described above is completely recursive, and dimension-invariant. Input `PIGrid`s and output regular grids of the same dimension are handled exactly as described above. The algorithm also behaves sensibly when the input `PIGrid` and the output regular mesh have different dimensions. Assuming that the target regular grid is m-dimensional and that the input `PIGrid` is n-dimensional with $m \geq n$, the `SMPL` operation consists of smoothly interpolating in the normal way onto an n-dimensional sub-grid of the output regular grid and extending by constant along the other axes. This can actually be very useful in practice, e.g. to generate "1.5D" and "2.5D" models. The other direction, where $m < n$, is also very useful and is handled in the obvious way, by outputting a slice in the 3-D case or a well in the 2-D case.

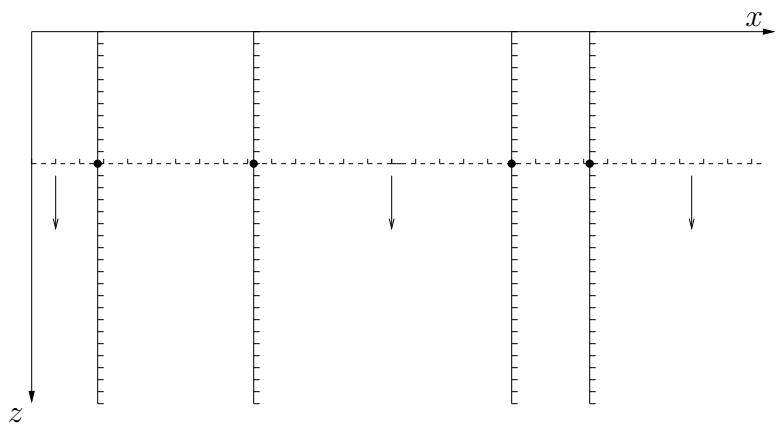

FIG. 9. Typical in-line section through the model of Figure 8

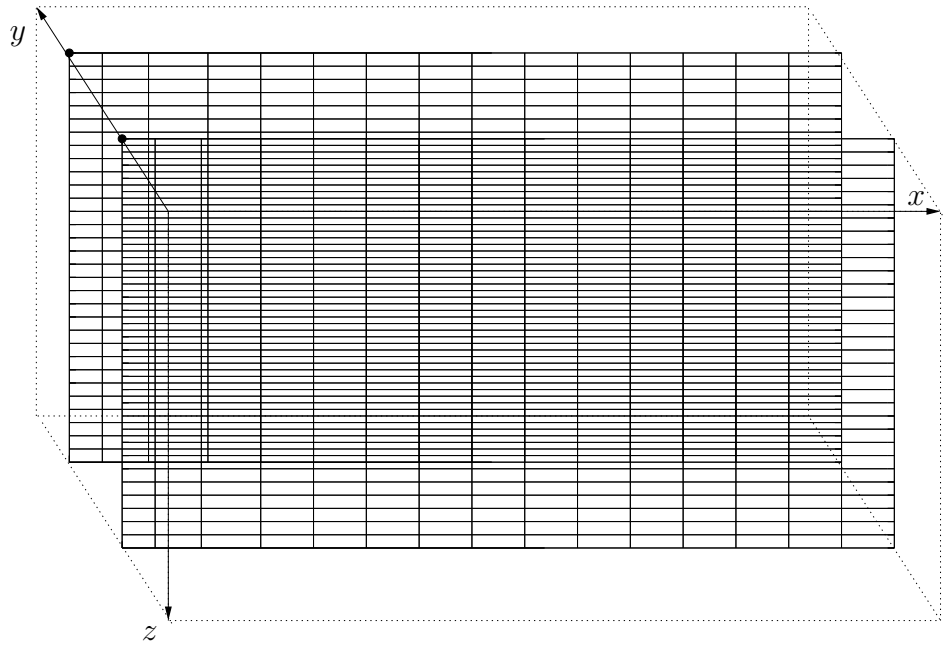

FIG. 10. Model obtained after two steps of `SMPL`

A 2-D EXAMPLE

Figure 11 (top panel) displays a 2-D (synthetic) velocity model, which, for the purpose of this example, we consider as the true model. A `PIGrid` is overlaid on top on the model. Each red line corresponds to a particular in-line coordinate, and each tick mark on these lines represents a node. This `PIGrid` consists of 52 nodes. The velocity prescribed at each of the nodes is precisely that at the corresponding location in the velocity model. Note that the slowly-varying part of the model should be very well represented on this `PIGrid`. On the other hand, there may not be enough nodes to represent accurately the high-velocity zone in the middle of the model.

The bottom panel in Figure 11 displays the velocity field obtained after smooth approximation on a regular grid (the parameters used are $n_x = 201$, $n_z = 101$, and $\Delta x = \Delta z = 10$m). The smoothing width used in this example is $h = 100$m along both depth and in-line directions. The overall result is quite acceptable. As expected, the structure in the middle of the model is not very well resolved, and suggests that more nodes be used. In fact, the result shows that variations along the depth and in-line directions can be well resolved with a small number of nodes, whereas variations not aligned in the axis directions require a higher number of nodes.

CONCLUSIONS

We have presented and described a sparse, hierarchical parametrization of the velocity which allows for user-defined placements of nodes. We also described an algorithm designed to exploit the special hierarchical structure to perform the smooth approximation efficiently, thereby allowing for user-controlled smoothness and guaranteeing that bounds are preserved (in the sense explained above).

The algorithm is **reversible**, in the sense that the adjoint operators of the SMPL operator and of the whole scheme have been implemented, thus allowing its use in gradient-based optimization techniques. That is, it is possible to start from the sparse representation, obtain a smooth model defined on a finely sampled regular grid, perform the optimization (step) on that grid, and get back to the original model space. In particular, the parametrization and the above algorithm have been used successfully in the context of NMO-based differential semblance optimization [Li, 2004].

REFERENCES

Beylkin, G. (1985). Imaging of discontinuities in the inverse scattering problem by inversion of a causal generalized Radon transform. *J. Math. Phys.*, 26:99–108.

Brandsberg-Dahl, S., deHoop, M., and Ursin, B. (2003). Seismic velocity analysis in the scattering angle/azimuth domain. *Geophysical Prospecting*, 51:295–314.

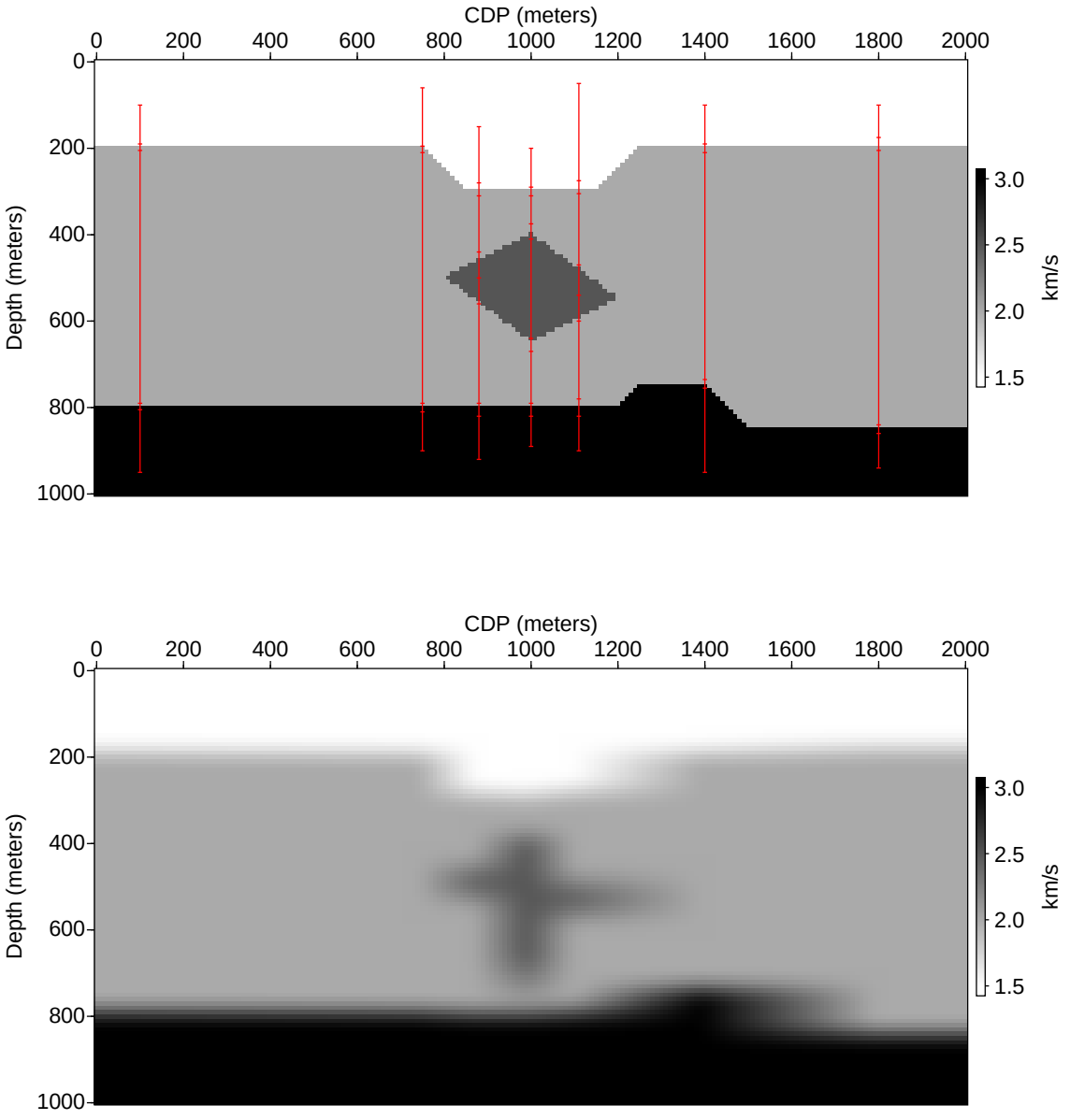

FIG. 11. Top panel: 2-D velocity model with the sparse model overlaid. Bottom panel: interpolated and smoothed model.

Li, J. (2004). Comparison of NMO-based conventional and differential semblance velocity analysis on several synthetic and field data sets. In *The Rice Inversion Project: Annual Report 2004*, Department of Computational and Applied Mathematics, Rice University, Houston, TX 77005-1892. (available electronically at `www.trip.caam.rice.edu`).

Padula, A., Scott, S., and Symes, W. (2004). The Standard Vector Library: a software framework for coupling complex simulations and optimization. In *The Rice Inversion Project: Annual Report 2004*, Department of Computational and Applied Mathematics, Rice University, Houston, TX 77005-1892. (available electronically at `www.trip.caam.rice.edu`).

Rakesh (1988). A linearized inverse problem for the wave equation. *Comm. on P.D.E.*, 13(5):573–601.

Trefethen, L. (1996). Finite difference and spectral methods for ordinary and partial differential equations. Available at `http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen`.

APPENDIX A

In this appendix, we reveal some of the implementation details and describe how to use the software in practice. The data structure representing `PIGrid`s as well as the algorithm performing the smooth approximation is implemented within the Standard Vector Library (`SVL`) framework. This library is a collection of C++ classes which realize in code the principal mathematical components of calculus in Hilbert space, and provides the framework for coupling complex simulations and optimization algorithms. For detailed descriptions of these classes and the design of `SVL`, see [Padula et al., 2004].

The implementation of `PIGrid`s mimics that of `Grid`s in `SVL`. `PIGrid`s contain the geometric description of the nodes, whereas `PIGridData`s contain the actual data values. `PIGrid`s are typically constructed using input files with a format specifically chosen to allow their recursive construction, regardless of their dimension. The input files are given the extension `.pig`. An example of such file in 3-D is:

ny	total number of cross-line coordinates
y_0	first cross-line coordinate
nx_0	total number of in-line coordinates at y_0
x_{00}	first in-line coordinate
nz_{00}	total number of depth coordinates at (y_0, x_{00})
z_{000} v_{000}	first (depth coordinate,velocity value) pair
0	asserts the above is a leaf
z_{001} v_{001}	second (depth coordinate,velocity value) pair
0	asserts the above is a leaf
.	
.	
sw 0 100	
sw 1 100	
sw 2 100	

Note that the smoothing widths (h in (4)) are specified at the bottom of the `.pig` file: `sw` is a keyword, the following integer specifies the dimension (0 for cross-line, 1 for in-line, 2 for depth) and the subsequent float is the actual smoothing width to be applied in that dimension. The many 0s appearing in the input files are essential to the recursive construction of `PIGrid`s, as they specify the nodes at the lowest levels of the tree (the leaves). In this way, the construction can be performed using the same code, whether in 1-D, 2-D or 3-D.

The unary functions objects `PIGridLoad` and `PIGridSave` are used to read from and write to file. For example, given a file "invel3d.pig", the code to instantiate a `PIGrid` and the associated `PIGridData` would look like:

```
PIGrid     pig(``inpig3d.pig'');       // creates the geometry
PIGridData pigdata(pig);               // allocates necessary storage
PIGridLoad pigload(``inpig3d.pig'');   // instantiates data loader

pigload(pigdata);                      // loads data in storage
```

A `PIGrid` object owns a reference to a (regular) `Grid` object which specifies the structure of the data contained by the leaves. For example, for the velocity model of Figure 1, the leaves are simple nodes, i.e. 0-dimensional `Grid` objects. However, after one step of the algorithm (see Figure 8), each leaf (in-line coordinates) in the tree has the structure of a 1-D `Grid`. Similarly, after two steps of the algorithm (see Figure 10), each leaf (cross-line coordinates) has the structure of a 2-D `Grid`. The `getSize()` method is a key member function of the `PIGrid` class, as it is used to specify the size of the `PIGridData` which holds the data associated with a `PIGrid` object. Note that the appropriate size is the total number of leaves times the size of the `Grid` object specifying their structure (note that a 0-dimensional grid has size 1, for it does contain a single scalar).

The binary function object `MasterSMPLFwdInterpFO` performs the recursive smooth approximation. It takes a `PIGridData` as input and outputs a corresponding `GridData`, assuming that the underlying `PIGrid` and `Grid` are compatible. As we mention before, the core of the algorithm is the `SMPL` operator which operates on 1-D arrays. The rest of the algorithm consists of re-arranging the data into 1-D arrays: the hierarchical data structure used to describe `PIGrid` makes this very easy to implement.