A dedicated Compression Scheme for Large Multidimensional Functions Visualization

M. Haefele
IRMA, UMR CNRS 7501
Université Louis Pasteur
Strasbourg, F-67084, France
haefele@math.u-strasbg.fr

F. Zara
LIRIS, UMR CNRS 5205
Université Lyon 1
Villeurbanne, F-69622, France
florence.zara@liris.cnrs.fr

G. Latu and J-M. Dischler
LSIIT, UMR CNRS 7005
Université Louis Pasteur
Illkirch, F-67412, France
name@lsiit.u-strasbg.fr

ABSTRACT

Large hyper-volume visualization is required by today physic and is still a research area in scientific visualization. Especially, the interactive exploration of datasets remains highly challenging as soon as the data size exceeds a certain threshold. We describe in this paper an hyperslicing-based interactive visualization technique designed to explore at real-time frame rates large hyper-volumetric 4D scalar fields (*i.e.* datasets beyond 16GB) defined on regular structured grids. The key issue consists in coupling the visualization with a new efficient data representation scheme, in such a way that it overcomes the different steps of scientific visualization, namely simulation post-processing, visualization preprocessing and finally interactive display. By introducing a hierarchical finite element representation, we show that our technique allows users to explore the full dataset at real-time frame-rates on low-end PCs. We demonstrate the effectiveness of our technique on the interactive exploration of 4D phases space particle beams, resulting from numerical semi-Lagrangian simulations.

Categories and Subject Descriptors

I.3.3 [**Computer Graphics**]: Picture/Image Generation; I.3.6 [**Computer Graphics**]: Methodology and Techniques; I.4.2 [**Image Processing and Computer Vision**]: Compression (Coding)

1. INTRODUCTION

In many scientific domains (physics, astronomy, biology, etc.), there are more and more numerical simulations generating huge amounts of complex numerical values, usually dense, multidimensional, multivariate and multi-scale time-varying. This constant evolution induces an increasing demand on efficient tools to help users to explore, analyze and visualize such large and densely sampled sets. Moreover, multidimensional dataset (with a dimension higher than three) are particularly challenging because of (1) huge amounts of memory requirements, and (2) no easy way to display such a complex information.

Many work in multidimensional visualization has already been driven. We can roughly classify them into two main categories according to the type of data they deal with: **multidimensional relation** visualization and **multidimensional function** visualization. Multidimensional relation visualization can be considered as database visualization, where a record of the database is considered as a multidimensional point. In [11], a survey of multidimensional relation visualization methods is presented.

In this paper, we are concerned with the second category: the multidimensional function visualization and in particular, scalar functions f defined as:

$$\begin{array}{rcl} f : \mathbb{R}^d & \rightarrow & \mathbb{R} \\ \mathbf{x} & \mapsto & f(\mathbf{x}), \end{array}$$

with $\mathbf{x} = (x_1, ..., x_d)$. Most visualization techniques have given up displaying the whole information at once and only display subsets of the information. With dedicated interfaces, users can then select these subsets interactively. In all cases, serious visualization problems arise when datasets are large because of core memory limitations and slow data accessibility. Since, these methods strongly resort on interactivity, real-time performance is a crucial issue.

The curse of the dimensionality arises when functions are defined on a regular discretization of the multidimensional space. Indeed, in this case the size S of the 4D volume dataset is given by $S = N^d.f$, with N the number of points that discretize one dimension, d the number of dimensions and f the memory required to store a single function value. To give an order of magnitude, for a discretization of $N = 128$ with a double precision float ($f = 8$ bytes), the memory required to store a single 2D, 4D and 6D dataset is respectively $S = 128$ KB, $S = 2$ GB and $S = 32$ TB. In this paper, we focus on 4D datasets ($d = 4$). Indeed, to our knowledge, no solution has been yet proposed to display and explore a full regular 4D scalar field that does not fit entirely into today low-end PCs memory (typically 32 GB dataset). An additional motivation for this work comes from plasma physic simulations [5, 8] which generate these kind of data.

In general, simulation post-processing, visualization preprocessing and management of large dataset for visualization are usually not considered together as a single problem. As a result, the global numerical cost of these steps can make

the visualization method unusable for practical purposes. Our approach consists in using a compression method to reduce the size of the numerical problem both for the simulation post-processing and for the multidimensional visualization. The visualization problem is then addressed by using a hyper-slicing technique [15]. The contribution of this paper is a new compressed representation of 4D volume datasets that is (1) well suited for dealing with huge amounts of data coming from numerical simulation, and (2) well adapted to hyper-slicing for efficient interactive visualization and data exploration.

The remaining parts of this paper are organized as follows. Section 2 presents some previous work on data compression and multidimensional data visualization. Section 3 deals with our compression method adapted to the visualization technique. Section 4 describes the visualization method and data structures used to store the compressed representation into core memory. Section 5 presents a performance evaluation of our method. Finally some concluding remarks and future work are given in Section 6.

2. PREVIOUS WORK

2.1 Multidimensional Visualization

In the framework of multidimensional function visualization, all proposed techniques have given up representing the whole information at one time and display only subsets of the information. In this kind of approach, the "grand tour" method [1] consists in building a sequence of 2D projections of a d-dimensional function. The difference between two consecutive projections is a little rotation of the projection plane. The result is then a 2D animation. The "hyperVolume method" [3] can be seen as an interactive "grand tour". As the user builds interactively a rotation with a smart interface, the projection is computed thanks to an efficient GPU implementation.

Other methods focus on the simultaneous display of several slices. The common point between these methods is to define slices with particular points $P = (p_1, p_2, \ldots, p_d)$ called **focus points**. A 2D slice $S_{a,b}$ which cuts the function in dimension a and b ($a, b \in [1..d]$), is then defined by:

$$\begin{array}{l} S_{a,b} : \mathbb{R}^2 \rightarrow \mathbb{R} \\ \quad (x_a, x_b) \mapsto f(\mathbf{x}), \\ \text{with } \mathbf{x} = (x_1, \ldots, x_a, \ldots, x_b, \ldots, x_d) \in \mathbb{R}^d, \\ \text{and } x_k = p_k, \forall k \in [1..d] \backslash \{a, b\}. \end{array} \quad (1)$$

This definition is easily extensible to 1D slice S_a and 3D slices $S_{a,b,c}$. Then, for all these methods, the data exploration is performed by refreshing the different slices as the user modifies the focus point. The main differences between them resides in the number and the type (1D, 2D or 3D) of slices displayed and the way the focus point is modified by the user. In the "Hyperslice method" [15] a $d \times d$ matrix of 1D and 2D slices is displayed. Slices are defined by a single focus point whose coordinates are displayed and can be modified on each slice. The "Hypercell method" [6] is an extension of the "Hyperslice method". The main differences consist in the possibility of displaying 3D slices and, instead of drawing all the possible slices, the user defines the ones of interest thanks to a smart workspace system.

In the context of the visualization of large 4D functions, whatever the visualization technique, the main challenge is to refresh the different slices interactively as the user manipulates the focus point. For this reason, we have chosen to focus our work on the improvement of data accessibility in the context of large dataset management. For our evaluation tests, we then implemented a traditional hyperslice approach [15], the key issue being the time spent to extract and reconstruct the desired slices.

2.2 Large Dataset Management

In general, large dataset management can be classified in three but non exclusive categories. First, **parallel solutions** divide the numerical problem by splitting data storage and data computation among different machines. Second, **out-of-core methods** process only the data subset of interest and leave the remaining data out of the core memory. Third, **compression methods** modify the information representation to reduce the size of the numerical problem. Although the two first categories bring solutions to the visualization problem, they leave the simulation post-processing whole. For this reason, we oriented our work on compression schemes. Then, two main constraints need to be satisfied. Firstly, our single computer execution constraint imposes on the compression scheme to reduce significantly the size of the data. Secondly, interactivity of the visualization method imposes on the compression scheme to provide an efficient random access to the entire dataset for hyperslice reconstruction.

Lossless compression algorithms ([18] for example) are not adapted to our concern. Indeed, the main property of these schemes is the exact recovery of the initial dataset after decompression. Thus, they provide very good compression ratio for text compression but they are definitely not adapted to scalar values compression.

Lossy compression algorithms can reach very high compression ratios. With this kind of methods, the original dataset is recovered with a bounded error ε. In the field of 3D, 3D+t and 4D visualization, one can make distinctions between **quantization methods** and **projection methods**. Quantization can be considered as lossy dictionary methods adapted to scalar values compression. The 3D+t vector quantization scheme used in [7] enables an efficient GPU implementation of the reconstruction algorithm. But the size of the resulting compressed data depends strongly on the initial dataset and, in some particular cases, this size can be bigger than the original dataset. Projection methods project in general the function (dataset) onto a specific base of a particular space function (typically L^2). So , the function is then represented by a linear combination of functions. Only coefficients of this linear combination have to be stored. Then, lossy compression comes into play when coefficients below a given threshold ε are discarded.

Visualization methods based on such compression use in general a three-step algorithm: (1) coefficients computation (projection) and suppression, (2) coefficients quantization and (3) encoding step. This last step enables memory savings but introduce an overhead in random accesses to the data. In [2, 9, 12], authors project their function onto different wavelet bases and propose memory representations of the encoded information to reduce this overhead. Instead

of using an encoding step, we propose a data structure (described in section 4) that still enables memory savings and provides very efficient random accesses. Projection methods used in [10, 14] consider the time dimension separately from the three space dimensions. But, this non-uniform compression leads to very inefficient random accesses in one specific dimension, an effect that we want to avoid. For this reason, we use an uniform 4D compression scheme.

Note that approaches presented in [13, 16] are close to ours because they really address the 4D problem at once. In particular, the hierarchical representation presented in [16] has some similarities with wavelets and multiresolution analysis, but the main motivation for the non use of wavelets was to build non overlapping basis functions within a subvolume, to reduce the cost of the reconstruction algorithm. Our compression scheme, described in the next section, stays in the framework of multiresolution analysis. This theoretical foundation allows us to propose not only the same reconstruction cost as [16] but also an efficient compression algorithm based on tensor product of 1D transformations.

To sum up, the proposed compression process suppresses quantization and encoding steps by using a dedicated sparse data structure. The uniform 4D compression scheme stays within strong theoretical foundations and allows linear complexity. Finally, locality make possible an efficient parallelization of our algorithms.

3. COMPRESSION SCHEME

3.1 Definition

Let's consider an interval $[a, b]$ subdivided in $N = 2^n$ sub-intervals $a = x_0 < x_1 < \ldots < x_N = b$ regularly spaced such that $h = x_{i+1} - x_i$. A function $f : [a, b] \to \mathbb{R}$ can be approximated by a function $f_h : [a, b] \to \mathbb{R}$ such that $f_h|_{I_i} \in \mathbb{P}^2$, with $\mathbb{P}^2$ the space spanned by two degree polynomials and

$$I_i = [x_i, x_{i+1}], \text{with } 0 \leqslant i < N, \tag{2}$$

one of the defined sub-intervals. Thus, on I_i, f_h is formulated by $ax^2 + bx + c$ and is completely defined by its values at the 3 points $x_i, x_{i+\frac{1}{2}} = \frac{x_i + x_{i+1}}{2}$ and x_{i+1}.

By setting the following conditions,

$$\begin{cases} f_h(x_i) = f(x_i) \text{ for } i \in [0..N], \\ f_h(x_{i+\frac{1}{2}}) = f(x_{i+\frac{1}{2}}) \text{ for } i \in [0..N-1], \\ f_h|_{[x_i, x_{i+1}]} \in \mathbb{P}^2, \end{cases} \tag{3}$$

the interpolating function f_h of f is uniquely defined and is a continuous piecewise polynomial function.

Then, the set of these functions f_h build a vector space V_h of $2N + 1$ dimensions (corresponding to the $(N + 1)$ x_i and the N $x_{i+\frac{1}{2}}$ points), with:

$$V_h = \{f_h \in C^0([a, b]), f_h|_{I_i} \in \mathbb{P}^2, \quad 0 \leqslant i \leqslant N\}. \tag{4}$$

So $f_h \in V_h$ can be decomposed onto a base of V_h, with:

$$f_h(x) = \sum_{i=0}^{2N} c_i \mathcal{L}_i(x). \tag{5}$$

Basis functions are then built by imposing

$$\mathcal{L}_i(x_{\frac{k}{2}}) = \delta_{i,k}, \quad i, k = 0, \ldots, 2N \tag{6}$$

with $\delta_{i,j}$ the Kronecker symbol. According to Equations (3) and (6), coefficients have the following property:

$$c_i = f(x_{\frac{i}{2}}). \tag{7}$$

Consequently, in this base, the coefficients of the decomposition are directly the value of the function and we have defined the well known $\mathbb{Q}^2$ finite elements [4].

In the context of numerical representation of functions, it is obvious to notice that the smaller h is, the better is the approximation f_h of f and the bigger is its numerical cost. So, we consider function f at different levels of resolution j. Recalling that h is the distance between two consecutive points of the initial discretization of f, the level of resolution j is defined by the distance $h_j = |x_{i+1} - x_i|$ between two consecutive points at this level (squares on figure 3). Steps h and h_j are linked by the following relation:

$$h_j = 2^{n-j} h, \text{with } 0 \leqslant j \leqslant n, \tag{8}$$

n being the finest level of resolution. The definition of I_i^j is almost the same as the previous one (2), with

$$I_i^j = [x_i, x_{i+1}], \text{with } 0 \leqslant i \leqslant 2^j - 1, \tag{9}$$

and its size corresponds to the distance h_j. Consequently, building a uniform $\mathbb{Q}^2$ approximation comes to take into account every point (*resp.* one point over two) of the initial discretization when considering level of resolution $j = n$ (*resp.* $j = n - 1$). But using a uniform base (figure 1) leads to errors where function f presents locally strong variations. To keep a good approximation on the whole domain, the finest resolution has to be kept, setting the compression ratio to one.

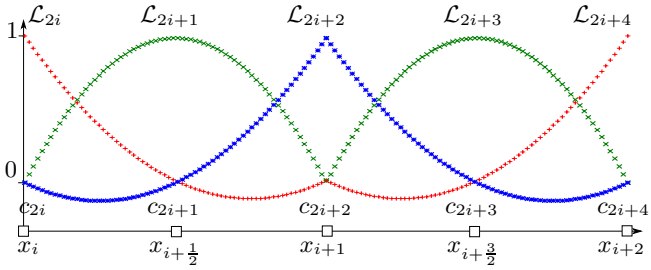


Figure 1: Uniform basis

Instead, we can build a hierarchical base [17]. Let's build this base by considering only two levels of refinement. Each sub-interval I_i^j can be hierarchically refined by adding two new basis functions to the existing ones. Then, we define $\mathcal{L}_{2i+\frac{1}{2}}$ by

$$\begin{cases} \mathcal{L}_{2i+\frac{1}{2}} \in \mathbb{P}^2([x_i, x_{i+\frac{1}{2}}]), \\ \mathcal{L}_{2i+\frac{1}{2}} = 0, \text{ if } x \notin [x_i, x_{i+\frac{1}{2}}], \\ \mathcal{L}_{2i+\frac{1}{2}}(x_i) = \mathcal{L}_{2i+\frac{1}{2}}(x_{i+\frac{1}{2}}) = 0, \\ \mathcal{L}_{2i+\frac{1}{2}}(x_{i+\frac{1}{4}}) = 1. \end{cases} \tag{10}$$

The other basis function $\mathcal{L}_{2i+\frac{3}{2}}$ is defined in the same manner on the interval $[x_{i+\frac{1}{2}}, x_{i+1}]$ (see Figure 2). As we have added two basis functions for each existing interval, the

dimension of the enriched vector space, called $\tilde{V}_h$, is then $4N+1$ and the approximation $\tilde{f}_h$ of f in $\tilde{V}_h$ is given by:

$$\tilde{f}_h(x) = \underbrace{\sum_{i=0}^{2N} c_i \mathcal{L}_i(x)}_{f_h(x)} + \underbrace{\sum_{i=0}^{2N-1} d_i \mathcal{L}_{i+\frac{1}{2}}(x)}_{g_h(x)}. \quad (11)$$

The computation of the coefficients d_i (often called details) consists simply in a linear combination of known function values (with $0 \leqslant i \leqslant j$):

$$d_{2i} = f(x_{i+\frac{1}{4}}) - \frac{3}{8}f(x_i) - \frac{3}{4}f(x_{i+\frac{1}{2}}) + \frac{1}{8}f(x_{i+1}), \quad (12)$$

$$d_{2i+1} = f(x_{i+\frac{3}{4}}) + \frac{1}{8}f(x_i) - \frac{3}{4}f(x_{i+\frac{1}{2}}) - \frac{3}{8}f(x_{i+1}) \quad (13)$$

Considering element $[x_i, x_{i+1}]$, Equation (11) can be evaluated at point $c = x_{i+\frac{1}{4}}$. Then this filter can be easily derived by using relation (7), the definition of basis functions (10) and the uniform discretization property.

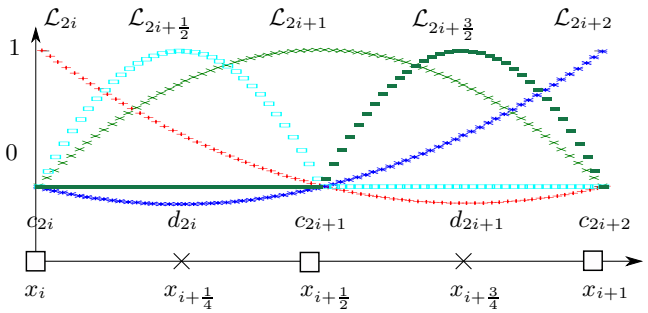


Figure 2: Hierarchical basis

Thanks to this hierarchical base (11), the approximation is now splitted into a coarse approximation f_h and a complement noted g_h. Considering a function f, function f_h can locally be accurate, leaving g_h close to zero. It implies small $|d_i|$ values that can be discarded by the compression algorithm without significant loss of accuracy.

This two levels decomposition can be generalized to a multi-levels decomposition. So we consider a coarse representation at a given level l (with $0 \leqslant l \leqslant n-1$) and a hierarchical representation from level l to n. We have now a multi-levels decomposition formula

$$f(x) = \sum_{i=0}^{2^{l+1}} c_i^l \mathcal{L}_i^l(x) + \sum_{j=l}^{n-1} \sum_{i=0}^{2^j} d_i^j \mathcal{L}_{i+\frac{1}{2}}^j(x). \quad (14)$$

Figure 3 shows a discretization with 9 points ($N = 8$) and how the hierarchical finite elements framework is mapped on it for a coarse level $l = 0$.

3.2 Compression Algorithm

The idea of the compression is to compute the different details d_i (projection of f onto $\tilde{V}_h$ space) and to keep every c_i^j coefficient where $|d_i^j| > \varepsilon$, with ε a given threshold. In practice, it comes to discard some values of the function which could be reconstructed with an acceptable error.

So the compression algorithm starts from the level of resolution $j = n-1$ and, for each coarser level until a given level l (with $0 \leqslant l \leqslant n-1$ and $0 \leqslant i \leqslant 2^l$), the algorithm computes the coefficients d_{2i}^j and d_{2i+1}^j thanks to Equations (12) and (13). Then, by keeping only $|d_i^j| > \varepsilon$, we build a compressed function f_c with a general wavelet representation:

$$f(x) \simeq f_c(x) = \sum_{i=0}^{2^l} c_i^l \mathcal{L}_i^l(x) + \sum_{j=l+1}^{n-1} \sum_{|d_i^j|>\varepsilon} d_i^j \mathcal{L}_{i+\frac{1}{2}}^j(x). \quad (15)$$

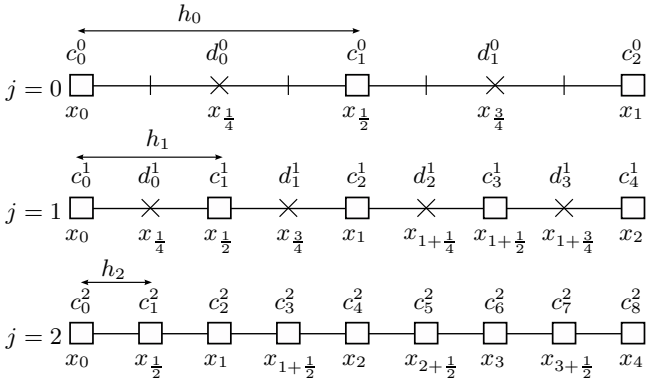


Figure 3: Domain decomposition according to resolution levels j (with $n = 3$)

The calculation of the contribution of d_i^j coefficients can lead to large computational efforts. The reduction of this cost motivates the approach in [16] which get out of wavelet representation. Instead, we propose to reduce this cost and stay within the wavelet framework by keeping a particular set of coefficients c_i^j such that

$$d_i^J < \varepsilon, \quad \forall J \geqslant j.$$

As a consequence, the compressed function f_c, built thanks to Algorithm 1, is given by:

$$f(x) \simeq f_c(x) = \sum_{c_i^j \in f_c} c_i^j \mathcal{L}_i^j(x). \quad (16)$$

Input: Discretized function f, compression threshold ϵ
Output: Compressed function f_c

$f_c = \emptyset$
for $j = n-1$ **to** $j \geqslant l$ **step** -1 **do**
  for $i = 0$ **to** $i \leqslant 2^j - 1$ **step** $+1$ **do**
    insert= 0
    for $k = 0, 1$ **do**
      Compute d_{2i+k}^j applying eq. (12),(13) **if** $|d_{2i+k}^j| > \epsilon$ **then**
        Add I_{2i+k}^{j+1} to f_c
        insert= 1
      end
    end
    if *insert* **then** Add I_i^j to f_c
  end
end

Algorithm 1: Compression algorithm

At the beginning of Algorithm 1, the compressed function f_c is empty. For each level of resolution, each element I_i^j is

considered to compute details d^j_{2i+k} (for $k = 0, 1$) belonging to this element. When a detail d^j_{2i+k} is greater than the threshold ε, all the coefficients belonging to the sub-element I^{j+1}_{2i+k} are inserted in f_c. It consists in inserting the three coefficients $c^{j+1}_{2(2i+k)+l}$ for $l = 0, 1, 2$. Note that, for tree structure purpose, if an element of level $j+1$ is inserted in f_c, his father I^j_i on level j must be inserted. It means that the three coefficients c^j_{2i+l} for $l = 0, 1, 2$ have to be inserted.

The 1D compression algorithm can easily be extended to d dimensions by using a tensor product of 1D transformations. A base of the d-dimensional space is built with all possible products of 1-dimensional basis functions. The main difference in the algorithm is the way elements are considered. In the general d-dimensional case, index i becomes a vector $\mathbf{i} = (i_1, i_2, \ldots, i_d)$ and an element $I^j_{\mathbf{i}}$ is defined by

$$I^j_{\mathbf{i}} = \prod_{p=1}^{d} I^j_{i_p}, \tag{17}$$

and contains 2^d sub-intervals $I^{j+1}_{2\mathbf{i}+\mathbf{k}}, \mathbf{k} \in [0..1]^d$.

3.3 Simulation Post-processing

Simulation post-processing or visualization pre-processing can be a big issue when dealing with large datasets. When simulation and visualization are considered separately, several reads/writes of the entire dataset are required, leading to high IO cost. Thus, the integration of our compression scheme into the simulation reduces the IO cost of the simulation and suppress completely the post-processing step.

For parallel simulations, this integration is not trivial. But, using our method, the compression of elements I^j_i is independent for a same level of resolution j. Thus, only one point on the boundary has to be shared between two elements. Consequently, except a communication step needed to share the boundaries, the compression process and the disk export of the compressed function can be distributed among parallel nodes. This compression scheme has been successfully integrated into our 4D parallel simulation [5, 8] and scales linearly with the number of processors.

4. 4D FUNCTION VISUALIZATION

4.1 Multidimensional Visualization Method

We have adapted the classical hyper-slicing technique described in [15] to address the 4D volume dataset visualization issue. This method represents a $d-$dimensional function with a $d \times d$ matrix of views. Each view can be a simple 1D curve plot or a 2D color map plot and displays a particular slice defined by a single focus point $P = (p_1, p_2, \ldots, p_d)$. The coordinates of the focus point are displayed and can be modified on each slice. By dragging it in the different views, the user modifies the displayed slices and has the feeling to explore the space as the slices refresh.

Figure 4 shows a screenshot of our application with a particle beam dataset. Instead of a matrix of views, only four 2D slices are displayed because for this application, the other slices don't make sense from a physical point of view. Details on the physic can be found in Section 5. According to notation (1), the top left view displays the 2D slice $S_{x,y} = f(x, y, p_{v_x}, p_{v_y})$.

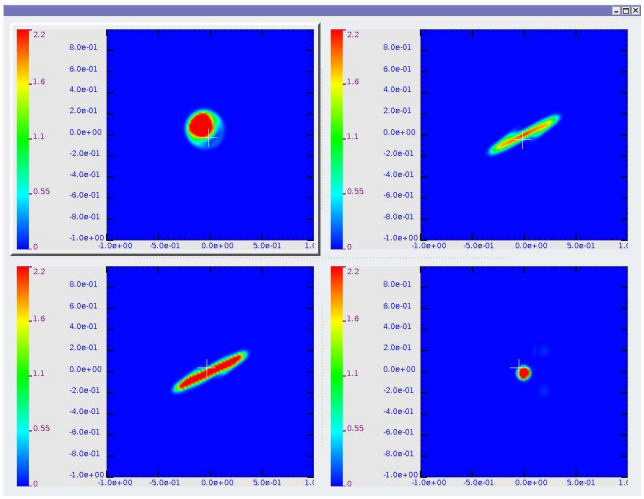


Figure 4: 4D distribution function from a particle beam simulation. From top to bottom and from left to right are presented the $(x, y), (x, v_x), (y, v_y)$ and (v_x, v_y) slices

Then, refreshing slices at interactive framerates is the critical point of this method. It has been overcome thanks to an efficient (both in storage and random accesses) sparse data structure (section 4.2) and thanks to an efficient slice reconstruction algorithm (section 4.3).

4.2 Data Structure

The memory representation of f_c is not trivial as it is not a regular array anymore. This mathematical object can be considered as a non structured mesh, composed of a set of elements, each element being a set of points. This approach leads to a point-connectivity representation which is very memory consuming in 4D (1 point = 4 indices and 1 element $= 3^4 = 81$ points).

Instead, we consider f_c as a cloud of points with implicit relations between them. To achieve efficient random accesses, we have designed a dedicated sparse data structure which considers two levels of refinement: one coarse and one fine. All points of a level of resolution j ($0 \leqslant j \leqslant l < n$) are stored in the coarse level noted C. As these points are simply an undersampling of the initial discretization, we use a regular 4D array to store f_c values. All other points belong to the fine level and are stored in cells according to their position. Each cell can be seen as a brick that fills the space between points of the coarse level. Figure 5(a) shows, for the 2D case, a partitioned domain with finite elements. Figure 5(b) presents only the points that match elements. It is a cloud of points representation which put them into two classes: coarse grid (disks) or fine cells (crosses). Figure 5(c) presents the memory representation. The coarse grid C is a regular array and fine cells are allocated to store all the other points. A reference to the allocated fine cells is kept in the regular array F.

A first memory saving is performed by allocating no memory when a cell is empty. Otherwise, two types of cell are used: **dense** and **sparse** cells. In dense cells, f_c values are stored in regular 4D arrays. They provide random ac-

cess in $O(1)$ (one indirection), but are clearly not optimal in memory when they contain only few points. In this case, a lexicographical sorted array of points is used, a point being four integer indices plus the f_c value. Thanks to the sorting, such sparse cells provide a $O(\log(M))$ random accesses (with M the number of points in the cell). Consequently, it is possible to optimize the memory consumption by using the appropriate cell type according to the bag-filling ratio of the cell. For example, in four dimensions with four bytes integer indices and four bytes floating point function value, it is cheaper to use a sparse cell when the filling ratio is lower than 50%, and a dense cell otherwise.

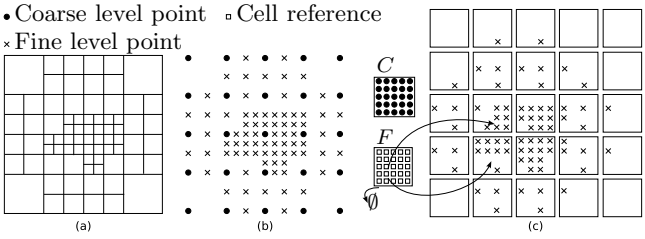


Figure 5: Sparse data structure

To sum up, this data structure can be seen as a dedicated hashmap implementation. Point indices play the role of the key which is associated to the f_c value. The hash function is a trivial shifting/masking operation. By optimizing its memory consumption, we are able to load completely the whole compressed function into the core memory. By keeping efficient random accesses, we ensure the interactive performances of the reconstruction algorithm.

4.3 Reconstruction Process

The slice reconstruction algorithm uses the finite element property saying that whatever the point considered inside an element, the value of the function at this point can be built with the function value on this element. So, the principle of the algorithm consists in computing the intersection between the 2D slice and the different 4D elements. Then the value of the function is computed on the 2D slice on the finest level of resolution in order to build the displayed colormap.

The data structure described in previous section does not store explicitly element information. But it exists implicitly because, thanks to the compression algorithm, every point required for it are present in f_c. And a sufficient condition to know if element $I_{\mathbf{i}}^j$ exists, is the presence of its center $C(I_{\mathbf{i}}^j)$ defined by

$$C(I_{\mathbf{i}}^j) = \frac{x_{\mathbf{i}} + x_{\mathbf{i}+1}}{2} = x_{\mathbf{i}} + \frac{h_j}{2}. \quad (18)$$

We can show that this point belongs only to element $I_{\mathbf{i}}^j$ when considering level of resolution j. For the sake of simplicity, Algorithm 2 presents the extraction of the 1D slice $S_2(x_2)$ from a 2D compressed function $f_c(\mathbf{x})$, with $\mathbf{x} = (x_1, x_2)$ and for the focus point $\mathbf{p} = (p_1, p_2)$.

At the beginning, all points of the slice are undefined. Then for each level of resolution j, centers $\mathbf{c} = (c_1, c_2)$ of all possible intersecting elements are computed. Once an intersecting element is found (presence of its center in f_c), the 1D Lagrange polynomial L is built. Its construction is described below. This polynomial is the approximation of f on the intersection between the slice and the domain of f: interval $[c_2 - h_j/2, c_2 + h_j/2]$. Then it is evaluated to build, on the whole intersection area, a discretization of the function at the finest level of resolution (step h). But values are actually written in the slice only at positions that are still undefined. If a value is already defined in the slice, it means that it has been computed using a finer level of resolution, *i.e.* with a better accuracy. Figure 6 presents the state of the slice for different levels j during Algorithm 2.

Input: Focus point $\mathbf{p}$, 2D compressed function f_c
Output: Slice S_2

Set S_2 to undefined ;
for $j = n - 1$ **to** $j \geqslant l$ **step** j -= 1 **do**
  $h_j = 2^{n-j}h$;
  $\mathbf{c} = \lfloor \mathbf{p}/h_j \rfloor h_j + h_j/2$;
  for $c_2 = h_j/2$ **to** $c_2 \leqslant 2^n$ **step** c_2 += h_j **do**
    if $f_c(\mathbf{c})$ *exists* **then**
      $m = c_2 - h_j/2$;
      $M = c_2 + h_j/2$;
      $L =$ `BuildPolynomial`$(f_c, j, \mathbf{c}, \mathbf{p})$;
      for $x_2 = m$ **to** $x_2 \leqslant M$ **step** x_2 += h **do**
        if $S_2(x_2)$ *is not defined* **then**
          $S_2(x_2) = L\left(\frac{2(x_2-m)}{M-m}\right)$;
        end
      end
    end
  end
end

Algorithm 2: Slice extraction

Thanks to tensor product construction of the 2D base, the approximation of function f on interval $[c_2 - h_j/2, c_2 + h_j/2]$ is a 1D Lagrange polynomial. By considering the reference element $[0, 2]$, we build the reference polynomial L

$$L(x) = f_0 + (f_1 - f_0)x + \frac{1}{2}(f_2 - 2f_1 + f_0)x(x - 1), \quad (19)$$

with $f_0 = f_c(p_1, c_2 - h_j/2), f_1 = f_c(p_1, c_2)$ and $f_2 = f_c(p_1, c_2 + h_j/2)$. But these three values may not exist in f_c. In this case, they have to be built by using a similar 1D polynomial, but in the other dimension. For example, on figure 6, while considering center $\mathbf{c}$ at level $j = 1$, the f_0 value doesn't exist in f_c. So the 1D Lagrange polynomial L_0 is built using $f_c(c_1 - h_j/2, c_2 - h_j/2), f_c(c_1, c_2 - h_j/2)$ and $f_c(c_1 + h_j/2, c_2 - h_j/2)$ and is evaluated at point p_1 to know f_0. If we note M the number of points in the slice, the complexity of this algorithm is in $O(M)$. As the amount of computations is not very important, the execution fastness depends on the random accesses efficiency to f_c values.

The principle of the extraction of a 2D slice $S_{1,2}$ from a 4D compressed function $f_c(x_1, x_2, x_3, x_4)$ is almost the same. Now the traversal of the possible intersecting elements is a 2D traversal. In the same manner, the traversal which fills the slice with the function value is also a 2D traversal. Finally, the polynomial computation can be performed by several 1D interpolations as in 2D.

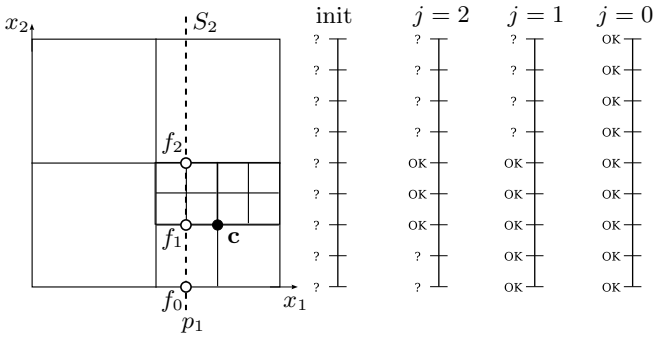
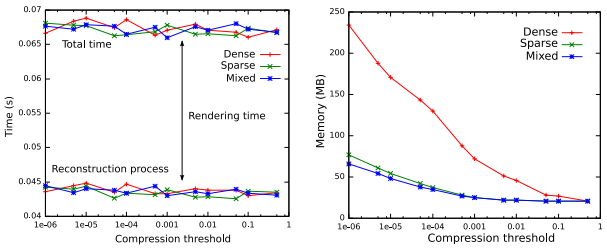


Figure 6: 1D slice extraction from a 2D function

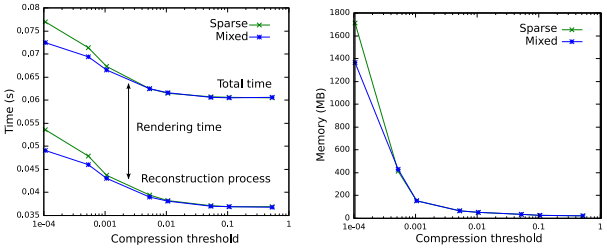
5. RESULTS

We mainly apply our technique to the interactive exploration of plasmas behavior. Plasma can be considered as the fourth state of material, which appears for huge temperature and pressure conditions (10^4K or more). These conditions are reached in different facilities, like fusion reactors (tokamak) and particle beam accelerators. Plasma is then modelled by a time-varying particle distribution function which lives in phase space, the product of the physical space with the velocities space. A Cartesian frame is used for particle beam and laser-material interaction. The 2D physical space (x, y) with the 2D velocities space (v_x, v_y) lead to a 4D distribution function $f(x, y, v_x, v_y)$. Numerical methods [5, 8] compute such 4D functions and integrate our compression scheme. Figure 4 (*resp.* 9) presents the visual result and figure 7 (*resp.* 8) presents the performances of our method for a particle beam (*resp.* plasma) simulation.



(a) Execution time (b) Memory required

Figure 7: Performance for particle beam dataset



(a) Execution time (b) Memory required

Figure 8: Performance for plasma dataset

The different tests are performed within a single thread on an Intel Pentium D (3.4 GHz, 2 MB cache). The evaluation of the visualization performances is carried out with a 256^4 grid, which represents 32GB array of double precision float. Three data structures that use different types of cells (cf Section 4.2) are compared. The "dense" test uses only dense cells, the "sparse" test, only sparse cells and the "mixed" test contains both types in order to optimize the memory consumption. Even with a good data size reduction, the required memory for plasma data is too large, this is the reason why we studied performances only for sparse and mixed data structure and only until threshold $\varepsilon = 10^{-4}$.

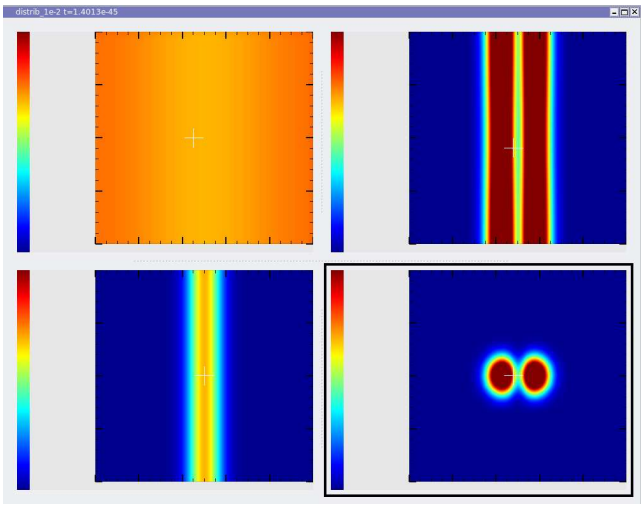


Figure 9: 4D function from plasma simulation. From top to bottom and from left to right are presented the $(x, y), (x, v_x), (y, v_y)$ and (v_x, v_y) slices

Figure 7(a) and 8(a) show the execution time of the visualization loop (focus point modification - slice computation - rendering) according to the compression threshold ε. As the displayed slices are always the same size, the rendering time is always the same ($\simeq 25ms$). The interactive frame rate of 10 fps is reached whatever the threshold and the type of cell. The increase of the execution time for low thresholds on Figure 8(a) comes from the large number of elements in the sparse data structure and its size leading to poor cache reuse. Figure 7(b) and 8(b) show the amount of memory required to load the entire compressed function into the core memory according to the compression threshold ε. As we expect, the mixed data structure presents the lowest memory consumption.

To sum up, according to a reasonable compression threshold $\varepsilon = 10^{-3}$, this compression scheme enables to reduce the data size by two orders of magnitude. And the sparse data structure provides random accesses efficient enough to make the reconstruction process interactive, giving to the user the illusion of the 4D exploration. So we can visualize 4D functions hundred times bigger with the same computer or visualize the same function on a computer hundred times smaller.

This technique has been recently implemented in the 5D non-linear GYrokinetic SEmi-LAgrangian code (GYSELA [8]). This code is developed at CEA Cadarache to perform simulations of ion turbulence in tokamak plasmas. Understanding turbulent transport in magnetized plasma is a subject of outmost importance for comprehending and optimizing experiments in the present fusion devices and also for designing future reactors. The development of such a 5D code (3D in real toroidal space (r, θ, φ) and 2D in ve-

locity space (the parallel velocity $v_\parallel$+ the magnetic momentum μ) is extremely challenging not only in terms of CPU time consumption and available memory size but also in terms of visualization. Indeed, one actual typical 5D distribution function needs 64 GB memory. Until now, only few 2D slices of this distribution function were exploited, typically poloidal cross-sections $S_{r,\theta}$ for two or three focus points. Now, this tool provides an interactive access to all possible slices without damaging the performances of the simulation code. Figure 10 shows a 4D distribution function $f(r, \theta, \varphi, v_\parallel)$ obtained by setting a specific μ in the 5D function. In particular, we notice on slice $(\theta, v_\parallel)$ the filamentation which appear in $v_\parallel$ direction in a developping turbulence phase.

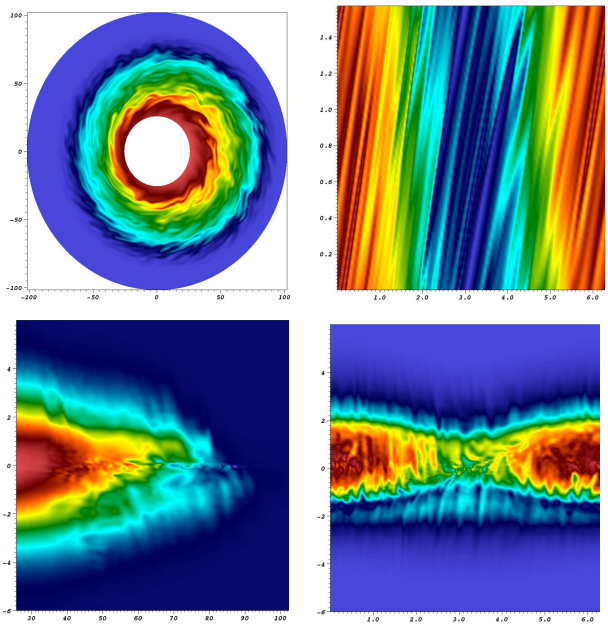


Figure 10: 4D function from gyrokinetic simulation. From top to bottom and from left to right are presented the $(r, \theta), (\theta, \varphi), (r, v_\parallel)$ and $(\theta, v_\parallel)$ slices

6. CONCLUSIONS AND FUTURE WORK

This paper presents a multidimensional visualization method for the exploration of 4D scalar fields. Its main contribution is the design of a dedicated compression scheme that both reduces significantly the size of the data and can be easily integrated into parallel simulation codes. The sparse data structure gives an efficient memory representation to store the compressed function and provides random accesses efficient enough to make the reconstruction process interactive. Finally, we provide an efficient tool for physicists which allows them to explore their data.

This work can be improved in both mathematical and computer science ways. Anisotropic compression scheme could allow good compression ratios even for functions which present strong variations over the whole domain. Indeed, compression ratio has to be improved for gyrokinetic datasets. An out-of-core extension can be designed thanks to the multiresolution representation of the data. These extensions would allow the visualization of even bigger 4D functions.

7. ACKNOWLEDGMENTS

This work has been done at LSIIT lab (Illkirch, France) and funded by Alsace Region, IRMA lab and INRIA. Thanks to Virginie Grandgirard from CEA Cadarache (France) and Eric Sonnendrucker from IRMA lab (Strasbourg, France) for gyrokinetic and particle beam datasets.

8. REFERENCES

[1] D. Asimov. The grand tour: a tool for viewing multidimensional data. *SIAM J. Sci. Stat. Comput.*, 6(1):128–143, 1985.

[2] C. Bajaj, I. Ihm, and S. Park. 3d rgb image compression for interactive applications. *ACM Trans. Graph.*, 20(1):10–38, 2001.

[3] C. Bajaj and V. Pascucci. Hypervolume visualization: A challenge in simplicity. Technical report, Austin, TX, USA, 1998.

[4] P. G. Ciarlet and J.-L. Lions, editors. *Handbook of numerical analysis. Vol. II.* North-Holland, 1991.

[5] N. Crouseilles, G. Latu, and E. Sonnendrücker. Hermite spline interpolation on patches for parallel Vlasov beam simulations. *Nuclear Instruments and Methods in Physics Research A*, 577:129–132, July 2007.

[6] S. R. dos Santos and K. W. Brodlie. Visualizing and investigating multidimensional functions. In *Symposium on Data Visualisation (VISSYM'02)*, pages 173–182, Aire-la-Ville, Switzerland, 2002. Eurographics Association.

[7] N. Fout, K.-L. Ma, and J. Ahrens. Time-varying, multivariate volume data reduction. In *ACM symposium on Applied computing (SAC'05)*, pages 1224–1230, New York, NY, USA, 2005. ACM Press.

[8] V. Grandgirard, Y. Sarazin, P. Angelino, A. Bottino, N. Crouseilles, G. Darmet, G. Dif-Pradalier, X. Garbet, P. Ghendrih, S. Jolliet, G. Latu, and E. Sonnendrücker. Global full-f gyrokinetic simulations of plasma turbulence. *Plasma Phys. Control. Fusion*, 49:B173–B182, 2007.

[9] M. H. Gross, L. Lippert, and O. G. Staadt. Compression methods for visualization. *Future Gener. Comput. Syst.*, 15(1):11–29, 1999.

[10] S. Guthe and W. Straßer. Real-time decompression and visualization of animated volume data. In *Conference on Visualization (VIS'01)*, pages 349–356, Washington, DC, USA, 2001. IEEE Computer Society.

[11] P. Hoffman and G. Grinstein. A survey of visualizations for high-dimensional data mining. *Information visualization in data mining and knowledge discovery*, pages 47–82, 2002.

[12] I. Ihm and S. Park. Wavelet-based 3D compression scheme for very large volume data. In *Graphics Interface*, pages 107–116, 1998.

[13] N. Neophytou and K. Mueller. Space-time points: 4d splatting on efficient grids. In *IEEE symposium on Volume visualization and graphics (VVS'02)*, pages 97–106, Piscataway, NJ, USA, 2002. IEEE Press.

[14] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In *Conference on Visualization (VIS'99)*, pages 371–377, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[15] J. van Wijk and R. van Liere. Hyperslice: visualization of scalar functions of many variables. In *Conference on Visualization (VIS'93)*, pages 119–125, 1993.

[16] J. Wilhelms and A. V. Gelder. Multi-dimensional trees for controlled volume rendering and compression. In *Symposium on Volume visualization (VVS'94)*, pages 27–34, New York, NY, USA, 1994. ACM Press.

[17] H. Yserentant. Hierarchical bases. In O'Malley and R. E., editors, *ICIAM 91*, pages 256–276, Washington, DC, 1992. SIAM, Philadelphia.

[18] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.