Transposing Data Volumes

May 10, 2016

Herman Jaramillo

1 Introduction

This is a short analysis of how to map a volume of data, either a memory buffer or disk file, referenced by a one dimensional pointer *p to the same set of data with a different sorting storage on memory. It is a generalization of the operation transposition in n dimensions. This idea was born as the need to modify the velocity field formatted in SEGY and with \( z \) (or \( t \)) as the fast direction to the same velocity field with \( x \) as the fast dimension to be read by Joe’s eikonal solver to be used in the Kirchhoff Prestack Depth migration code. The applications of this idea are many. Not only, transpositions are useful in general but, for example, sorting seismic data is a routine task. The usual meaning of sorting of seismic data is along all directions except the fast direction. That is, the primary and secondary access keys are either line, bin, shot, receiver or offset...etc, but never depth (\( z \)) or time (\( t \)). However when displaying time (or depth) slices it is useful then to transpose the order of the data so that (\( t \)) or (\( z \)) are the slow directions.

An important application of sorting data in any direction is that some filters which applied as multidimensional are complicated could be performed by cascading them as one-dimensional filters. For example, if we want to re-grid a velocity model so that the output grid dimensions are \( dx_o, dy_o, dz_o \), while the input corresponding dimensions are \( dx_i, dy_i, dz_i \) and we have an algorithm to do this along the fast dimension (say \( z \)), we could apply that algorithm three times after sorting the data so that the fast direction is \( z, y \) and \( x \). The first time we do not have to sort anything, because it is already sorted with \( z \) as the fast dimension.
We should consider this problem in n dimensions. A complete seismic data volume is five dimensional (including offset and azimuth), the travel time tables for prestack Kirchhoff depth migration are five dimensional sets. If we define new attributes we could increase dimensionality so we should better know how to attack the general n-dimensional problem.

I am sure that this analysis is already considered somewhere in books (I would look first at Donald E. Knuth books but I do not have them. I will leave this as a future task), however I will do my work from scratch.

2 The problem

To better understand the problem I will post it for three dimensions and later on I will generalize it to n dimensions.

Let us assume that we receive a volume of data from disk (or memory) into a pointer *p. This volume of data is indexed as \([i_x][i_y][i_z]\) meaning that \(i_z\) is the index for fast dimension and \(i_x\) is the index for the slow direction. We want to read the data into a pointer *q where the order of indexing is \([i_z][i_y][i_x]\) so that now \(i_x\) index the fast dimension while \(i_z\) index the slow dimension. We also will assume that the array sizes along the three dimensions \((x, y, z)\) are respectively \((n_x, n_y, n_z)\). We will index things from zero as in C-syntax.

On the operation
\[ q[\text{index}] = p[k], \]

where, as we will show below, \( k = i_z + i_y \times n_z + i_x \times (n_y \times n_z) \) we want to find out what is “index”.

3 The solution

I will show two different approaches to the solution of this problem.

3.1 the computer scientist approach

The piece of code to perform what we want is (in C-syntax pseudo-code)

```c
for(ix =0; ix<nx; ix++) /* reading each cross-line */
{
```

for(iy=0; iy<ny; iy++)
{
    read(seismic trace on pointer *p); /* along time/depth direction */
    for(iz=0; iz<nz; iz++)
    {
        index = 0;
        q[index]=p[iz+60]; /* header has 60 words */
    }
}

To answer what goes into the interrogation mark "?" we have to know how the data should be sorted out. As stated above, let us assume that the data will be allocated in memory as \( [i_z][i_y][i_x] \) indicating that the fast dimension is \( x \) and the slow dimension is \( z \). So that if we want to read that data into a pointer *r*, the code to read the data would be written as

\[
\begin{align*}
\text{for(iz =0; iz<nz; iz++ )} & \quad /* reading each cross-line, for each depth */ \\
& \quad \{ \\
& \quad \quad \text{for(iy=0; iy<ny; iy++)} \\
& \quad \quad \quad \{ \\
& \quad \quad \quad \quad \text{read(data in q by chunks of nx samples);} /* along x direction */ \\
& \quad \quad \quad \quad \text{for(ix=0; ix<nx; ix++)} \\
& \quad \quad \quad \quad \quad \{ \\
& \quad \quad \quad \quad \quad \quad \text{index = ix + iy*nx + iz*(nx*ny);} \\
& \quad \quad \quad \quad \quad \quad \text{r[index]=q[ix];} \\
& \quad \quad \quad \} \\
& \quad \} \\
& \}
\end{align*}
\]

Let me explain how did I get the index value for this problem. Going from the outer to the inner loop. We have to transverse on the outer loop \( i_z + 1 \) steps but the last step is not finished yet, so that we will count only \( i_z \) complete steps, each complete steps requires \( nx \times ny \) steps (for the two inner loops). We have transversed \( i_y + 1 \) steps but the last step is not complete yet, so that we will count \( i_y \) complete steps along this direction, each with \( n_x \) cycles; finally we will have for the inner loop \( i_x + 1 \) steps after looping through the \( i_x \) element. The total amounts to \( i_x + i_y \times n_x + i_z \times (n_x \times n_y) \) as indicated by the index.

So the code for reading as \( [i_x][i_y][i_z] \) and writing as \( [i_z][i_y][i_x] \) is given by using this index in the first flow. That is
for(ix =0; ix<nx; ix++) /* reading each cross-line */
{
    for(iy=0; iy<ny; iy++)
    {
        read(seismic trace on pointer *p); /* along time/depth direction */
        for(iz=0; iz<nz; iz++)
        {
            index = ix + iy*nx + iz*(nx*ny);
            q[index]=p[iz+60]; /* header has 60 words */
        }
    }
}

3.2 the geometrist solution

A geometrist will see this problem as accessing a volume with preferred directions. For example if we want to access the cell of a volume gridded as $[i_z][i_y][i_x]$ we will think of moving along the $x$ direction each time (from left to right) and once we get to the right extreme we should move toward the $y$ direction (from front to back), and once this is done toward the bottom (top to bottom). After being on the coordinate $[i_z][i_y][i_x]$ we have moved through $i_z$ complete phases of the cube, each phase with areas of $n_x * n_y$, and in the last phase, we have moved $i_y$ rows, each row with $n_x$ elements, finally we have to complete the $i_x + 1$ elements of the last move with the index $i_x$. This will gives us again the index of $i_x + i_y * n_x + i_z * (n_x * n_y)$.

So the important thing is how the output will go, no how the input is being read. Here is an easy to remember (mnemotechnic rule) trick:

First write the indexes from fast to slow. For example, in 4D write $i_w \ i_x \ i_y \ i_z$. Then multiply the slowest index $i_z$ by the complementary orthogonal cube volume (all dimensions except the $z$ dimension), that is $n_w * n_x * n_y$, the following slowest index $i_y$ by the complementary orthogonal area above it $n_w * n_x$, and the next index $i_x$ by the length of its complementary orthogonal segment length $n_w$ and add the last free index $i_w$. This would result in

$$\text{index} = i_w + i_x * n_w + i_y * (n_w * n_x) + i_z * (n_w * n_x * n_y) \quad (1)$$

In general if we have a hyper-volume grid indexed by $i_1, i_2, \ldots, i_m$ with dimension sizes of $n_1, n_2, \ldots, n_m$ respectively. The indexes are sorted from
fast to slow, and we want to store this hyper-volume grid in memory following a different sorting, as for example \( j_1, j_2, \ldots, j_m \) (from fast to slow) with dimensions mapped as \( n_{j_1}, n_{j_2}, \ldots, n_{j_m} \) then the general index for reading the data is given by

\[
\text{index} = \sum_{k=1}^{m} j_k \prod_{l=0}^{k-1} n_{j_l}
\]

(2)

where \( n_{j_0} = 1 \).

4 Conclusions

I derived a general formula for transposing volumes of data in any dimension. This formula might be in many places on the literature, however I decided to treat it as unknown (in fact it was unknown to me). While the formula is useful for many tasks, my first objective is something to get me moving on the development of 3D prestack Kirchhoff depth migration. Joe’s eikonal solver reads data with \( x \) as the fast dimension, while standard SEGY seismic data comes with \( z \) (or \( t \)) as the fast dimension.