The new FST compression algorithms

Yves Chartier – 2nd version – May 2006[1]

Introduction

This document describes the compression algorithms recently introduced in the RPN standard files.

Floating point numbers inside RPN standard files are stored internally as a stream of unsigned integers through a process called **quantization**. The size of each quantized values is NBITS. NBITS can vary from 1 to 24 bits, 12 and 16 bits being the values used the most often. This yields a compression ratio of 2.0 for the R16 data type (an R16 field is composed of 16-bit unsigned integers) and 2.67 for the R12 (12-bit unsigned integers) data type.

The formula used to transform the real values into integers is

$$\mathbf{INT = 2^{nbits} * (REAL - min) / range}$$

where range is 2^N, N being the nearest exponent such that 2^N > (max – min). This process is lossy, there is an irreversible loss of precision occuring in that transformation. However, the amount of error induced by this transformation is generally within acceptable levels, often much smaller that typical observation errors.

The demonstration of the technique will be done with the help of a practical example. The 2 following fields, a sea level pressure 0 hr forecast and a land-sea mask extracted from a GEM regional run, will be referred to in the text.

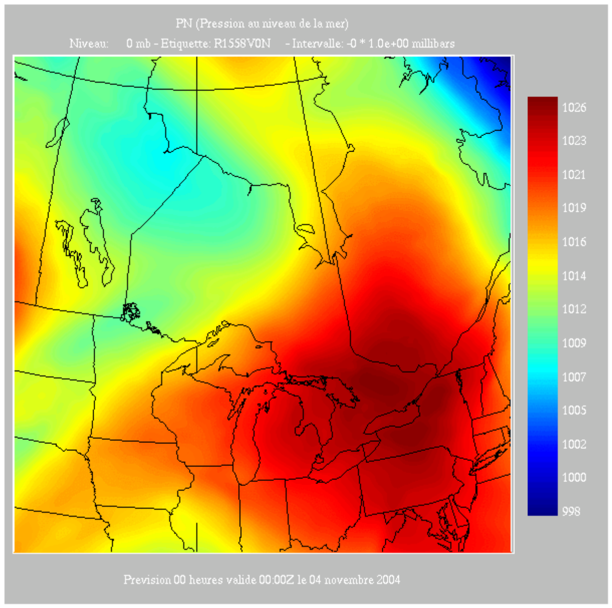

PN (Pression au niveau de la mer)
Niveau: 0 mb - Etiquette: R1558V0N - Intervalle: -0 * 1.0e+00 millibars
1026
1023
1021
1019
1016
1014
1012
1009
1007
1005
1002
1000
998
Prevision 00 heures valide 00:00Z le 04 novembre 2004

Figure 1

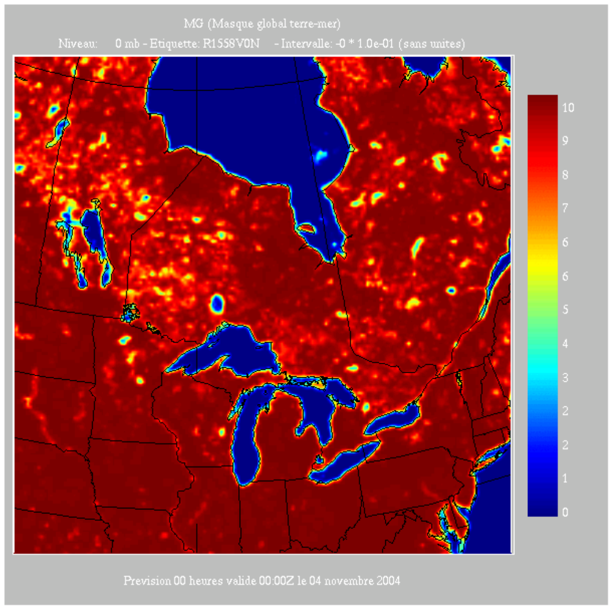

MG (Masque global terre-mer)
Niveau: 0 mb - Etiquette: R1558V0N - Intervalle: -0 * 1.0e-01 (sans unites)
10
9
8
7
6
6
5
4
3
2
2
1
0
Prevision 00 heures valide 00:00Z le 04 novembre 2004

Figure 2

Currently there are 2 compression methods implemented, one that we will call "**minimum**", and the other that we will call "**Lorenzo predictor**", which replaces the "**bi-cubic**" method. Both methods use the bi-dimensional property of the data structures.

1 (The 1st version version of this document was produced in November 2004)

The minimum tile method

This method segments the field in 5x5 tiles, as shown in the following diagram.

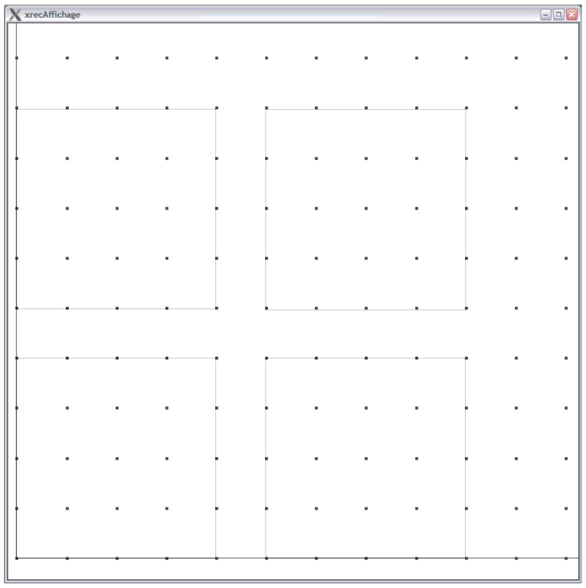


Figure 3 shows the first 5x5 values of the PN (sea level pressure field). Some statistics about the field are shown in the right column.

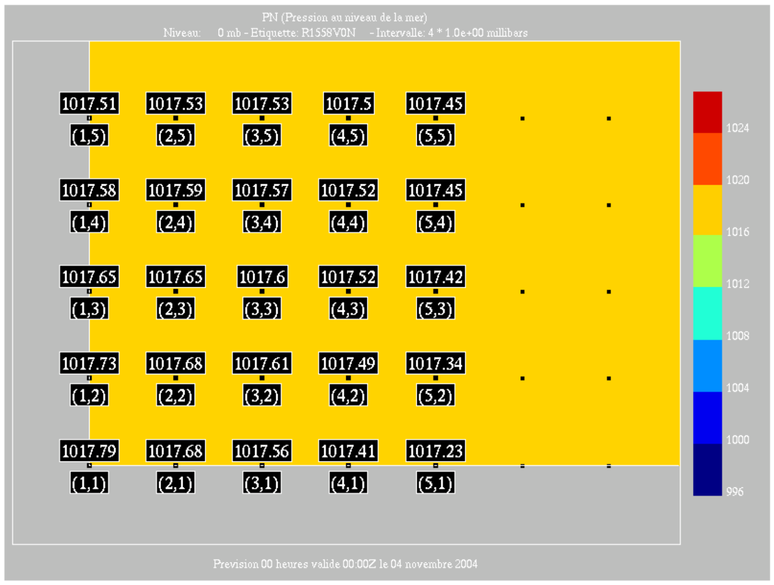


Figure 3

Max	1026.71301
Min	997.797913
Delta	28.9151
Normalized Range	32
nbits	16

The 1st point of the PN field has for value 1017.78619.

This value would be encoded as
2^{16} * (1017.78619 – 997.79791) / 32.0 = 40936

Here are the values of the first 5x5 grid.

1017,58	1017,59	1017,57	1017,52	1017,45
1017,65	1017,65	1017,60	1017,52	1017,42
1017,73	1017,68	1017,61	1017,49	1017,34
1017,79	1017,68	1017,56	1017,41	1017,23

when encoded in 16 bits these values become

40373	40415	40417	40340	40254
40515	40537	40498	40389	40240
40665	40659	40551	40659	40551
40812	40727	40565	40331	40565
40936	40726	40474	40166	39804

The minimum technique consists in encoding the minimum value of tile, and in each cell the difference between the actual value and the minimum. If the difference is small in the tile, then we need less bits to encode the tile, thus saving space.

This is the same tile encoded with the minimum method

569	611	613	536	450
711	733	694	585	436
861	855	747	855	747
1008	923	761	527	761
1132	922	670	362	0

The minimum value is located at point (5,1), the maximum at (1,1). The tile range is 1132, implying that 11 bits are required to encode this tile (2^{11} = 2048). The bitstream is composed of the following elements :

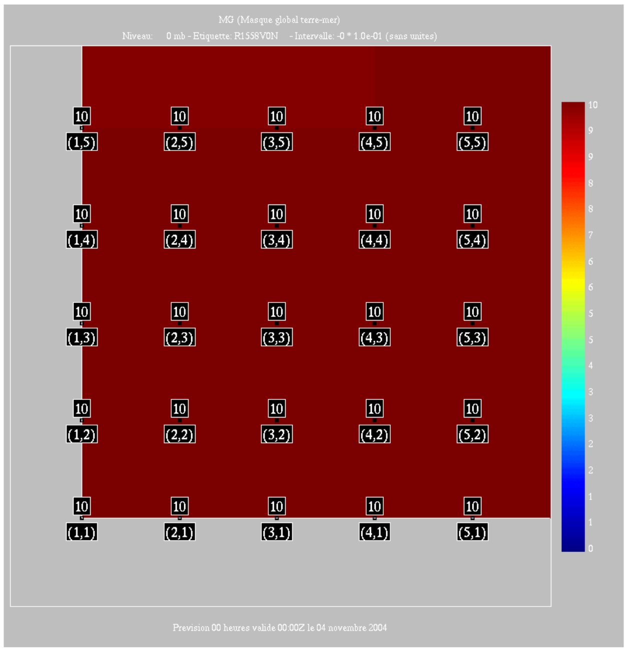
- nbits_needed (the number of bits needed, from 0 to 15),
- the minimum value of the tile
- the 25 (5x5) encoded differences between the quantized values and the min.

This tile would be encoded that way.

word	0				1			
bits	0	8	16	24	0	8	16	24
token	nbits	Minimum		Fld(1,1)	Fld(2,1)		Fld(3,1)	Fld(4,1) ...

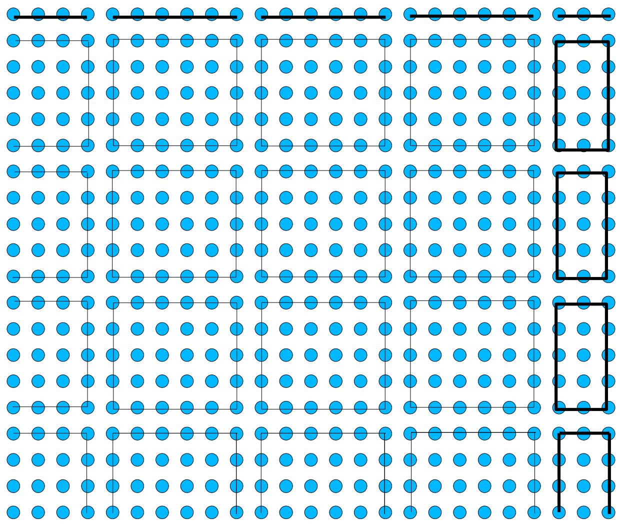
The total space taken by this 5 x 5 tile is
nbits_needed (4) + minimum value (16) + 25 tokens * 11 bits/token = 4 + 16 + 11*25 = 295 bits.

The original stream was taking 25 * 16 bits = 400 bits. So we save 105 bits, or a compression ratio of 400 / 295 = 1.36. The encoding of the second tile starts exactly on the next bit closing the firs 5 x 5 tile.

Consider the same 1st 5x5 tile from the MG field. All the values are identical. In that case, the stream is composed of : • nbits_needed (here 0), • the minimum value of the tile, which is in fact a constant. In that case we need only 20 bits : 4 for **nbits_needed**, and 16 for the constant token. In that case the savings are 20:1, 20 bits instead of 400.	

Adjustment of grid boundaries

The size of the subcells is adjusted to the limit of the grid, as shown in the following diagram.



Additional comments and observations

Why a 5x5 cell size ?

The optimal cell size is a compromise between the overhead required for each cell (**nbits_needed** and **local_minimum**) and the reduction of the range of observed values within a cell.

The smaller the cell, the smaller the variation of the field inside the cell, the smaller the number of bits needed to represent the variation, yielding more compression. But then we need more cells, adding a fixed per-cell overhead.

Empirical tests have shown that 5x5 the cell size that gives the best average compression.

Worst Case Scenario :

All the 5x5 cells have values that cover the minimum and maximum range of the field. In that situation we observe an expansion of the field, due to the overhead of having to include for each cell the **nbits_needed** and **local_minimum** fields. This overhead is (4 + nbits) per 5x5 cell. For a 16 bit 1000x1000 field, the overhead would be 1000000 / 25 * (4+16) = 800000 bits. So the field would take 16,800,000 bits instead of 16,000,000, an increase of 5 %.

Best Case Scenario :

A field with constant values. For each 5x5 cell we need only (4 + nbits) bits. For the same 1000x1000 field, the size taken by the field would be 800,000 bits, yielding a compression ratio of 16,000,000 / 800,000 = 20.0.

Average scenario :

There is no such thing since the fields vary so much ! Currently, a 24 hour 12-bit regional GEM prog in ETA coordinates taking 587 megabytes (1055 fields of dimension 576x641) can be compressed to 278 megs (a 2.11 compression ratio). The globel GEM prog in ETA coordinates valid at the same time taking 89 megs compresses to 54 megs (a 1.64 compression ratio).

The NBITS_NEEDED parameter :

This field takes 4 bits, and hence has a range from 0-15. To accomodate the cases when 16-bits are needed, we decided to use 16 bits when NBITS_NEEDED is 15 or 16.

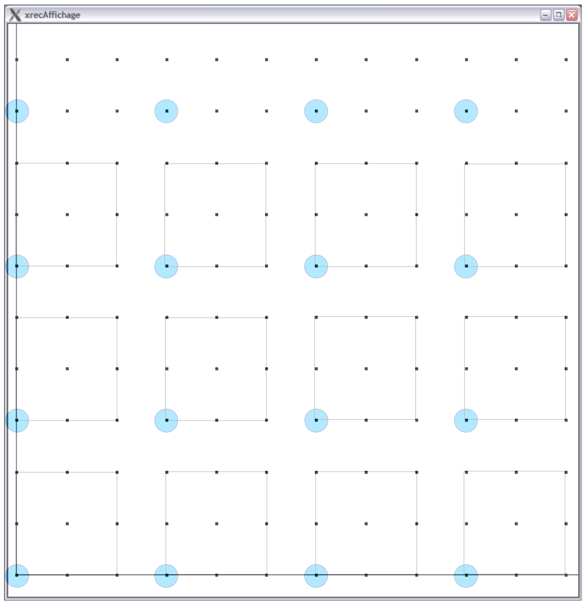
The bi-cubic sample tile method

This methode is not used anymore, since the “Lorenzo predictor” gives 20% more compression and runs almost as fast as the minimum tile method.

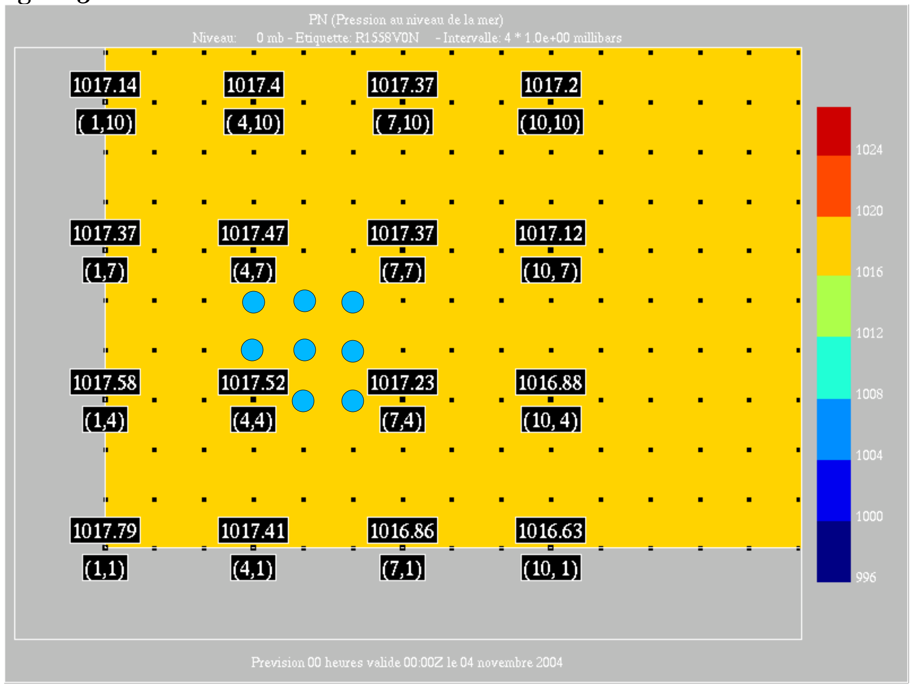
However a fair amount of work has been done in the design of this algorithm, so I think it is fair to keep this part of the documentation. There is also the fact that some legacy datasets have been saved using that algorithm, so this information can be useful in accessing this data.

In that method we decompose the field in 3x3 sub-grids. The 1st point of each sub-grid is a part of a coarse grid from which a bi-cubic prediction of the values of the other points will be computed. What we need to keep in order to reconstruct the field is the coarse grid and the prediction errors. If the prediction errors are small, then we can save space.

By doing many tests involving different step sizes, empirical observations have shown that the optimal step size for the bi-cubic interpolator is 3.



Here is a sample view of the coarse grid needed to predict values for the 8 points of the subcell shown in blue for the sea level pressure field shown in Figure 3.



So given the coarse 4x4 grid, we want to interpolate values for the zone in red.

40077			40286			40092			39581
			🟥	🟥	🟥				
			🟥	🟥	🟥				
40515			40389	🟥	🟥	39794			39075
40936			40166			39046			38564

After the bi-cubic interpolation, we have

Original values			Predicted values			Predicted errors		
40286	40265	40202	40339	40277	40174	-53	-12	28
40340	40254	40133	40377	40269	40119	-37	-15	14
40389	40240	40032	40389	40226	40023	0	14	9

The bitstream for the bi-cubic sample method is theoritically split in two parts

- the coarse grid (nbits * ni_coarse * njcoarse)
- the differences (prediction errors)

coarse_grid	prediction_errors

The prediction errors part can be expressed as a stream of subcells

- nbits_needed
- sub-tile (8 * nbits_needed) tokens

coarse_grid	nbits_needed (1..3,1..3)	err(2,1)	err(3,1)	err(1,2)	err(2,2)	err(3,2)	err(1,3)	err(2,3)	err(3,3)	nbits_needed (4..6,1..3)	Err(5,1)	Err(6,1)...

The size of the coarse grid depend on the grid geometry, but is roughly 1/9 of the original

- ni_coarse=INT((ni+4)/3)
- nj_coarse=INT((nj+4)/3)

nbits_needed is the number of bits needed to encode the differences. In the example shown above, we have a maximum error of 53. So we need 6 bits per token to cover this range ($2^6 = 64$), plus one more bit for the sign (which unfortunately we cannot avoid), which yields 7 bits.

The total space in bits taken by this tile is 4 + 8 * 7 = 60 bits. The original stream would have taken 8 * 16 = 128 bits, so this is a saving of 108/60 = 2.13. Please note that the worst error in that predicted tile is 0.13 % . The average prediction error for the whole tile is 0.06 %.

The following could be a valid encoding of the tile.

word	0					1				
bits	0	8	16	24		0	8	16	24	
token	nbits	fld(2,1)	fld(3,1)	fld(1,2)	fld(2,2)	fld(2,2)	fld(1,3)	fld(2,3)	fld(3,3)	nbits

The reader can take note that we do not use a reference token (as in the minimum method). It has been empirically observed that the errors are on average evenly distributed around 0.

When all the prediction errors are 0, only the field "nbits_needed" is encoded, yielding a compression ratio of 128 / 4 = 32.

As a measure of protection we have added a supplementary field called **nbits_req_container**, positioned between the coarse grid and the error streams. This field is an insurance policy against the possible overshoot or undershoots caused by the bi-cubic method.

If the original stream is 16-bits, then it is possible for the prediction errors produced by the bi-cubic method to be greater than 65535.

Here is a worst case scenario

65535	0	0	65535
0	65535	65535	0
0	65535	65535	0
65535	0	0	65535

Numerical analysis shows that the maximum value one can find is about ±81626, expanding the needed range to 17 bits.
So this is why we introduced this 3-bit field, which can contain only two values : 4 or 5 bits.

The final bitstream lies down as follows

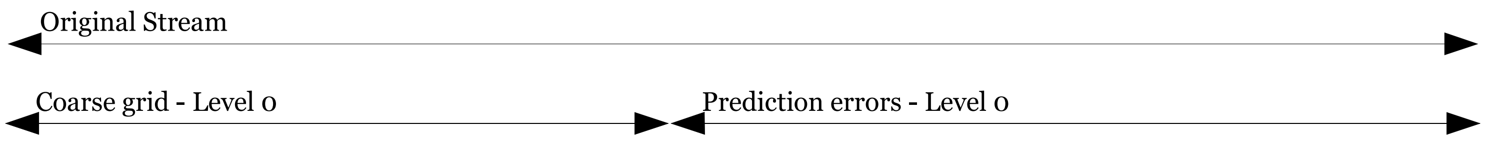
coarse_grid	sizeof(nbits)	prediction_errors

An additional optimization

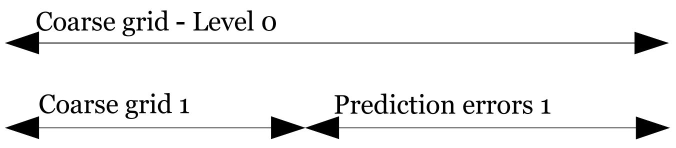
Consider the following data structure

coarse_grid	prediction_errors (….)

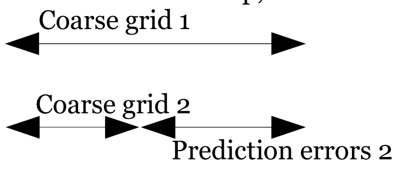
If the prediction errors are small, then the size taken by the coarse grid becomes a significant part of the total compressed data. It has been found that re-applying the same predictor-corrector scheme on the coarse grid could yield some additional compression. Here is a graphical outline of the method.



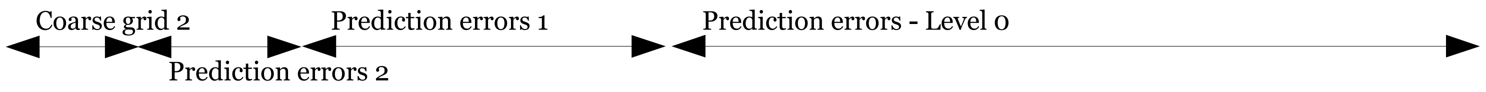
At level 1, the 0 level coarse grid is decomposed into another coarse grid and prediction errors, that we will call "coarse grid 1" and "Prediction errors 1".



And the the final step, where "coarse grid 1" is decomposed into "coarse grid 2" and "prediction errors 2".



The final stream is assembled this way.



It was decided that two-more level of compression would be added to the streams. We stopped to 2 levels because the potential savings were infinitesimal.

The following table outlines the savings possible when we have a compression ratio of 2 between each coarse grid level.

	% of multi-level compressed field								
Level	***Coarse grid 2***	***Errors 2***	***Coarse grid 1***	***Errors 1***	***Coarse grid 0***	***Errors 0***	***% of level 0 compressed field***	***Compression ratio obtained***	***% of original field***
0					22.2 %	77.8 %	100.0 %	2,00	50,00%
1			2.78%	9.72%		87.5%	88.89 %	2,25	44,44%
2	0.31 %	1.10 %		9.86%		88.73%	87.65 %	2,28	43,83%

The following table outlines the savings possible when we have a compression ratio of 4 between each coarse grid level.

Level	***Coarse grid 2***	***Errors 2***	***Coarse grid 1***	***Errors 1***	***Coarse grid 0***	***Errors 0***	***% of level 0 compressed field***	***Compression ratio obtained***	***% of original field***
0					44.4 %	55.6 %	100.0 %	4	25,00%
1			7.41%	9.26%		83.3 %	66.67 %	6	16,67%
2	0.87%	1.09 %		9.80%		88,24%	62,96%	6.35	15.74%

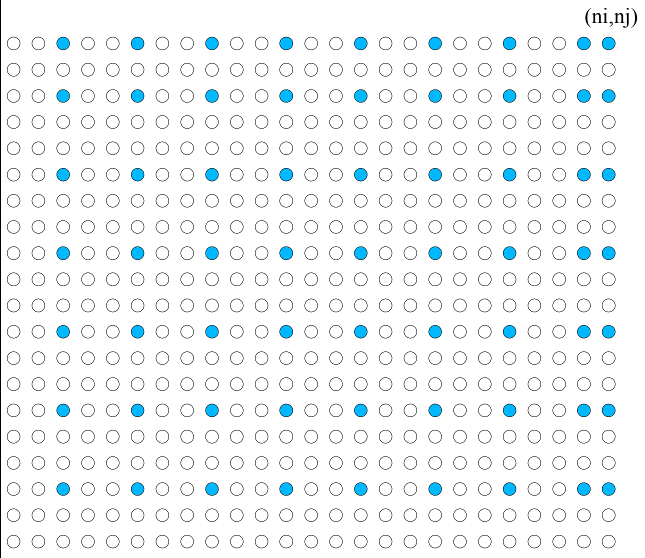
At level 0, we decompose the original stream into a coarse grid and prediction errors.
In the above diagrams, we neglected for simplicity the field “nbits_needed” between the coarse grid and the prediction errors. The final compression scheme looks like this.

Here is the final data layout (not drawn to scale).

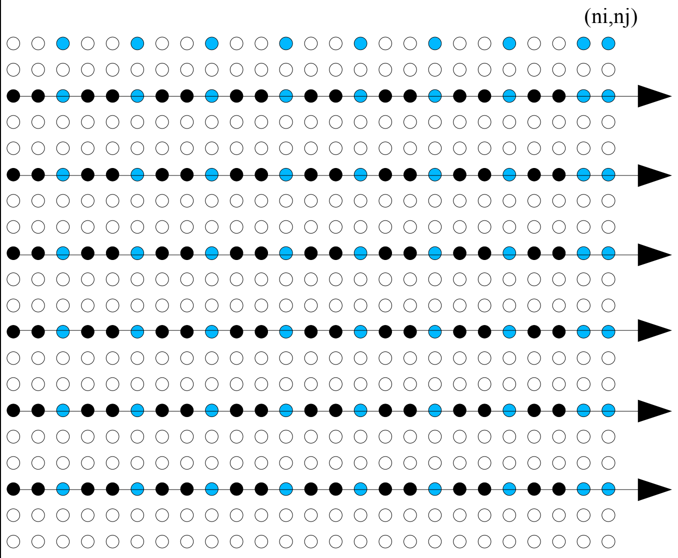
coarse_grid					sizeof(nbits_needed)	prediction_errors (....)
coarse_grid1			nbits_needed1	pred_errors1	sizeof(nbits_needed)	prediction_errors (....)
coarse_grid2	nbits_needed2	pred_errors2	nbits_needed1	pred_errors1	sizeof(nbits_needed)	prediction_errors (....)

Interpolation steps

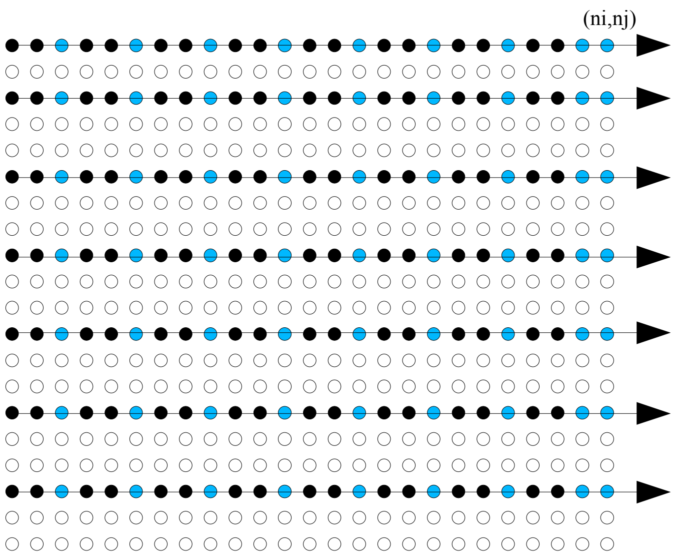
We will show graphically the steps used in the prediction of the inner points.



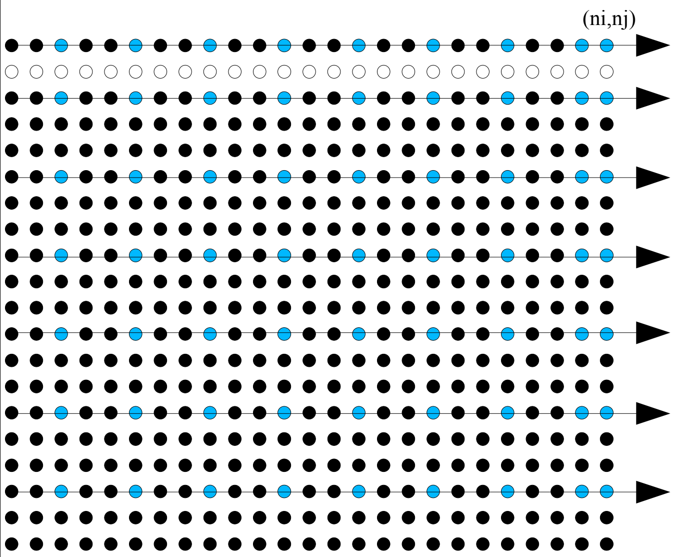
In the following example, we have ajus_x=2, ajus_y = 1 (ajus_x = (ni-1) % 3, ajus_y = (nj-1) % 3). The points in blue define our coarse grid. Remark the spacing of the points in the right 2 columns and the upper 3 rows.



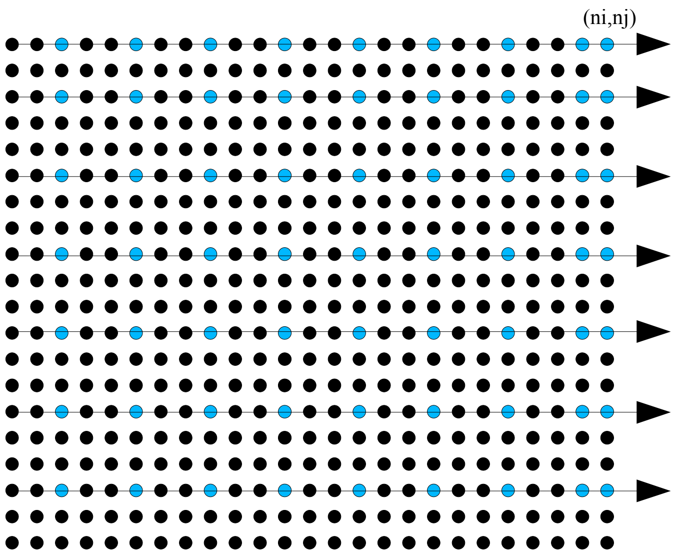
We start the interpolation on the inner grid points aligned with the coarse grid y axis. We stop to the last row that is a multiple of 3 points away from the 1st row.



We then interpolate the top row by horizontal bicubic interpolation.



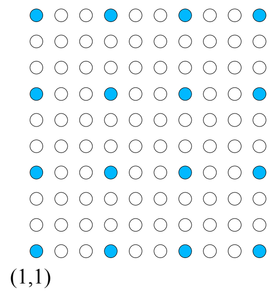
We interpolate all the inner points except the but last row.



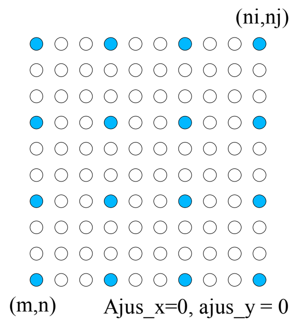
We finally interpolate the next-to-last row.

Adjustment of grid boundaries

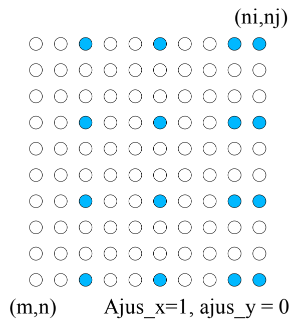
Independently of grid boundaries, the coarse grids always start at point (1,1) and increase by steps of 3 in every axis.



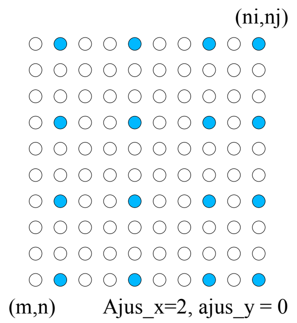
The following picture shows all the possible combinations of grid boundaries with a tile step of 3. For the last before column where ajus_x = 2 the predicted value of z(ni-1,j) = 0.5*(z(ni-2,j)+z(ni,j)). For the last before row where ajus_y = 2, the predicted value of z(i,nj-1) = 0.5*(z(i,nj-2)+z(i,nj))



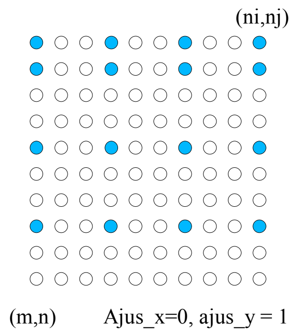
Ajus_x=0, ajus_y = 0



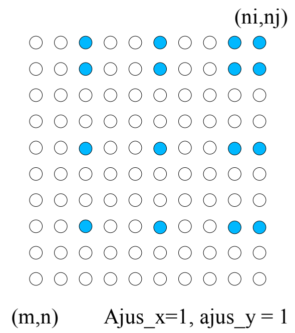
Ajus_x=1, ajus_y = 0



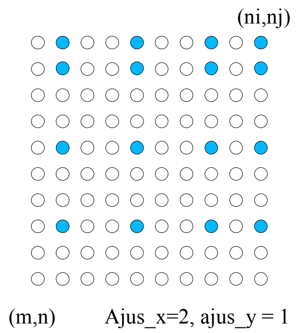
Ajus_x=2, ajus_y = 0



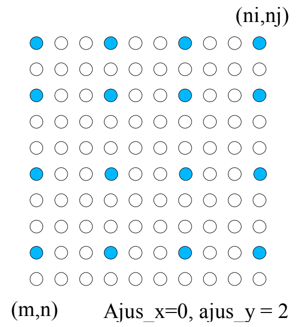
Ajus_x=0, ajus_y = 1



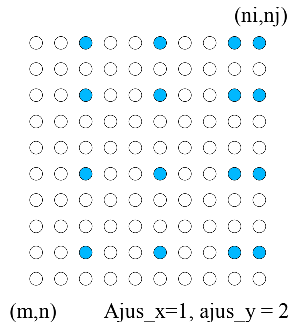
Ajus_x=1, ajus_y = 1



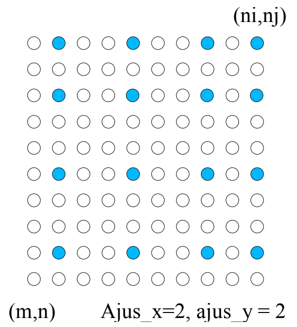
Ajus_x=2, ajus_y = 1



Ajus_x=0, ajus_y = 2



Ajus_x=1, ajus_y = 2



Ajus_x=2, ajus_y = 2

Stabilization of the bi-cubic predictor

The following code is what is used in standard cubic interpolation algorithms, where 0.0 <= dx <= 1.0.

```
              dx
|---------|----*----|---------|----
z1        z2        z3        z4

parameter (one = 1.0D0)
parameter (three = 3.0D0)
parameter (six = 6.0D0)
parameter (sixth = one/six)
parameter (third = one/three)
real*8 cubic, dx,dy,z1,z2,z3,z4

cubic(z1,z2,z3,z4,dx)=((((z4-z1)*sixth + 0.5*(z2-z3))*dx  + 0.5*(z1+z3)-z2)*dx
                         + z3-sixth*z4-0.5*z2-third*z1)*dx+z2
```

The above formulation can lead to round-off differences when executed on different computers or compilers, because there are 4 divisions in this formulation (dx = 1/step or 2/step, third = 1/3, sixth = 1/6).

In order to avoid this potential problem, we reformulated the cubic function like this

```
parameter (fac1 = 108.0D0) ! 12 * 3 * 3
parameter (fac2 = 1944.0D0) ! 72 * 3 * 3 * 3
parameter (unsurfac2 = 1.0D0/fac2) ! 1 / (72 * 3 * 3 * 3)
icubic(z1,z2,z3,z4,dx)=z2+(dx*(6*(dx*(2*(dx*((z4-z1)+3*(z2-z3)))+18*((z1+z3)-2*z2)))+
                      fac1*(6*z3-z4-3*z2-2*z1)))* unsurfac2
```

This formulation is valid only for a step factor of 3. In that formulation dx is an integer (1 or 2). Since we are working in double precision, and that maximum tokens have a value of 65535, the results of the multiplications are always exact. We have only one division left, mult/1944. The only round-off problem that remains is when mod(mult, 1944) = 972, in which the division result is of course 0.5 but can be computed as 0.499999, which when truncated will be off by 1 digit.

This potential problem was solved by adding the number 0.5001 to the result of the function.

The following table shows the effect of adding this factor to the result when mod(mult,1944) = 971, 972 or 973. Since the numerator cannot be greater than 81626, at 64-bit arithmetic we have ample precision to represent the numbers shown in the table.

Integer value	971	972	973
Real value	0.49948560	0.50000000	0.50051440
value + 0.5001	0.99958560	1.0001	1.00061440
Truncated value	0	1	1

Additional comments

Worst Case Scenario :

All the 3x3 cells have values that cover the minimum and maximum range of the field. In that situation we observe an expansion of the field, due to the addition of the 5-bit **nbits_needed** field + 18 bits per interpolated cell. Because of the three-level prediction scheme, it is difficult to define a general formula. For a 1000x1000 field 16-bit field, we can compute the following table

	bits								
Level	*Coarse grid 2*	*Errors 2*	*Coarse grid 1*	*Errors 1*	*Coarse grid 0*	*Errors 0*	*Total size*	*Compression ratio obtained*	*% of original field*
0					1784896	16621844	0,00%	0,87	115,04%
1			200704	1869056		16621844	18691604	0,86	116.82 %
2	23104	215156		1869056		16621844	18729160	0,85	117.06 %

In that worst case scenario, the compressed field takes 17% more space than the original.

Best Case Scenario :

A field with constant values. Let's compute the savings for the same field as above. Note that the size of the prediction errors is the nicoarse * njcoarse * nbits_needed.

	bits								
Level	*Coarse grid 2*	*Errors 2*	*Coarse grid 1*	*Errors 1*	*Coarse grid 0*	*Errors 0*	*Total size*	*Compression ratio obtained*	*% of original field*
0					1784896	446224	2231120	7,17	13,90%
1			200704	50176		446224	697104	22,95	4,40%
2	23104	5776		50176		446224	0	30,46	3,30%

We can see here that the compression ratio reached is 30,46, which is better than the theoritical maximum of 20.0 that we can get from the minimum method.

Average scenario :

This method yields on average an increase of 30 % in the compression ratio, compared to the minimum method.

The NBITS_NEEDED parameter :

As in the minimum method, this field takes 4 bits, and hence has a range from 0-15. To accomodate the cases when 16-bits are needed, we decided to use 16 bits when NBITS_NEEDED is 15 or 16.

Minimum grid size :

To work the bicubic interpolation scheme needs at a minimum a 4x4 grid.

If we reverse the grid size computation scheme, we get the following coarse grid sizes.

Level 2	4 x 4
Level 1	8 x 8
Level 0	20 x 20
Original	56 x 56

For the bi-cubic interpolation scheme to work, we need that the minimum dimension of one of the two grid axes to be 56.

A practical example

Let's take the PN example shown in Figure 3.
The grid size is 181x181. From these values, the following quantities are derived :

ni	181
nicoarse	61
nicoarse_level1	21
nicoarse_level2	8

The size of the multi-level compressed field is 263848 bits (compared to an original size of 524176 (181*181*16 bits).

The “level 0” prediction errors amount for 223200 bits.
The “level 1” prediction errors amount for 34656 bits (including the 3 bits required for the **nbits_required** field).
The “level 2” compression gives a size of 5992 bits : 1024 for the coarse grid (8 * 8 * 16), 3 for the number of bits required to encode the maximum errors (**nbits_required**, can be 4 or 5) and 4965 for the prediction errors.

So the compressed data stream would be structured this way (cell size not drawn to scale)

coarse_grid2	sizeof(nbits_needed2)	pred_errors2	sizeof(nbits_needed1)	pred_errors1	sizeof(nbits_needed)	prediction_errors (....)
5992 (8x8)	3	4944	3	34653	3	223200
0.39%		1.87 %	3	13.1 %	3	84.6 %

If we had done only 1 level of compression, the stream size would have been

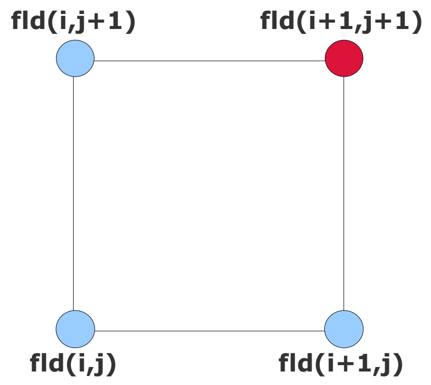
coarse grid	59536 (61 * 61 * 16 bits / token)
nbits_needed	3
prediction_errors	223200
Total	**282739**

So by using a 3-level compression scheme, we saved 282739 - 263848 = 18891 bits. This yields a compression ratio of 1.99 instead of 1.85, an increase of 6.7%

The Lorenzo predictor tile method

The Lorenzo predictor (also called the parallelogram predictor) has been introduced to the author in October 2003, during a seminar about compression held during the Visualisation 2003 conference.[2]

The Lorenzo prediction algorithm is deceiptively simple, it is illustrated here.



fld(i+1,j+1) = fld(i,j+1) + fld(i+1,j) – fld(i,j)

This method has a degree 1 accuracy in 2-dimensions, degree 2 in 3-D and degree 3 in 4-D.

The value of point (i+1,j+1) is predicted from its 3 neighbors, and what is stored is the prediction error.

Here is what we get with the numeric example introduced in the mininum tile method.

The original integer stream

40373	40415	40417	40340	40254
40515	40537	40498	40389	40240
40665	40659	40551	40659	40551
40812	40727	40565	40331	40565
40936	40726	40474	40166	39804

The predicted stream with the Lorenzo algorithm (the values in gray are seed values and are not predicted)

40373	40395	40376	40308	40191
40515	40509	40429	40606	40281
40665	40580	40497	40317	40893
40812	40602	40475	40257	39969
40936	40726	40474	40166	39804

The prediction errors

0	-20	-41	-32	-63
0	-28	-69	217	41
0	-79	-54	-342	342
0	-125	-90	-74	-596
0	0	0	0	0

2 Look on the web for papers : http://www.gvu.gatech.edu/~jarek/papers/Lorenzo.pdf

If we look at the first 3x3 predicted cells

-28	-69	217
-79	-54	-342
-125	-90	-74

We see that the range of values goes from -342 to 217. To compute how many bits are needed, we take the maximum of the absolute value of the errors and we add theoritical sign bit. In that case the maximum is 342, thus we need 9 bits (2^9=512) + 1 bit for the sign = 10 bits.

word	0					1			
bits	0		8	16	24	0	8	16	24
token	nbits	Fld(2,2)		Fld(3,2)	Fld(4,2)	Fld(2,3)		Fld(3,3)	Fld(4,3)

In this example the Lorenzo predictor does not seem to perform much better than the minimum tile method but in practice it proved to be the best of the three methods implemented. One big advantage of this method is that it has near zero overhead : only ni + nj – 1 original values need to be kept to recreate the original stream. By comparison, the overhead of the two other methods is of order n^2. It also requires only 2 integer mathematical operations (compared to the 125 flops of the bi-cubic method).

As for the bi-cubic method, it happens that the prediction errors can be greater than 16 bits. Here is a worst case scenario. For this sample of original values

65535	0
0	65535

the predicted value here would be 65535 + 65535 – 0 = 131070, and the prediction error is 0-131070 = -131070, requiring 18 bits to be encoded. In that case, the “nbits_required” field would be set to 5 instead of the default of 4.

So, as for the bi-cubic method, the first element of the bit stream is a 3-bit token containing the number of bits required for the NBITS field : 4 or 5. Then follow the 1st row of the field, then the 1st column of the field (minus the 1st element), and then the prediction errors, broken in in sets of 3x3 cells, each containing the minimum number of bits required to encode the errors and and the prediction errors.

nbits_req	Orig values (1..ni,1)	Orig values (1,2..nj)	nbits_needed (2..4,2..4)	Err(2,2)	err(3,2)	err(4,2)	Err(2,3)	Err(3,3)	err(4,3)	Err(2,4)	Err(3,4)	Err(4,4)	Next cell (....)

Additional comments

Why a 3x3 cell size ?

As for the other two methods the optimal cell size is a compromise between the overhead required for each set of cells (**nbits_needed**) and the reduction of the range of observed values within a cell.

The smaller the cell, the smaller the variation of the field inside the cell, the smaller the number of bits needed to represent the variation, yielding more compression. But then we need more cells, adding a fixed per-cell overhead.

Empirical tests have shown that 3x3 the cell size that gives the best average compression, although there was not a big difference with a 4x4.

Worst Case Scenario :

All the 3x3 cells have values that cover the maximum possible range of the field, which is nbits + 2 (nbits + 1 because of the error + 1 sign bit). In that situation we observe an expansion of the field, due to the increased number of bits and the overhead of having to include for each 3x3 set the **nbits_needed** field. This overhead is 4 or 5 bits per 3x3 cell. For a 16 bit 1000x1000 field, the overhead would be roughly 1000000 * 4 / 9 + 16*1000000*(18/16) ≈ 18445000 bits, an increase of 15 %.

Whenever the compressed buffer is larger than the original, an error code is returned to the calling routine so that the field is written to the file in its original state (ie uncompressed).

Best Case Scenario :

The size if the field in bits is roughly[3] nbits*(ni + nj – 1) + 4*(ni*nj/9). If nbits = 16, ni=nj=1000, then the compressed stream size is of the order of 476500 bits, yielding a compression ratio of 16000000/476500=33.5. If nbits=12, this ratio becomes 12000000/476500 = 25.2.

Average scenario :

This method yields on average an increase of 15 % in the compression ratio, compared to the bicubic method, and 40 % compared to the minimum method.

The NBITS_NEEDED parameter :

As in the minimum method, this field takes 4 bits, and hence has a range from 0-15. To accomodate the cases when 16-bits are needed, we decided to use 16 bits when NBITS_NEEDED is 15 or 16. As mentioned above, this field can be expanded to 5 bits if the prediction errors exceed the 16-bit range (ie greater than 65535).

3 The exact value depends on the size of the grid. There might be an extra cell added in each dimension.