1 File Format and Decompression for PFSS Fieldline Data

This section describes the file format and decompression algorithm for the fieldline data. The data is saved in the FITS file format [1] and compressed with RAR [2].

To decompress the data do these steps

1. decompress the data with RAR.
2. Read the FITS file. The content of the FITS file is described in subsection 1.1.
3. Decode Bytes with the algorithm specified in section 1.2.
4. Split up the data into the corresponding fieldlines. This step is described in-depth in subsection 1.1.1.
5. Decode predictive coding described in section 1.3.

1.1 FITS Content

The FITS file is composed of the header and the body. The content of the header and body are described in table 1 and 2. NOTE: The content of the (table 2) is encoded. The raw Bytes have to be retrieved and decoded before use. The Byte decoding is described in subsection 1.2.

Name	Datatype	Content Description
b0	Float	Latitude to Earth
l0	Float	Longitude to Earth
Q1	Float	Quantization Coefficient 1
Q2	Float	Quantization Coefficient 2
Q3	Float	Quantization Coefficient 3

Table 1: Header of the FITS File

Name	Encoding	Content Description
LENGTH	Adaptive Unsigned	Length of each fieldline
X	Adaptive Signed	X channel of all fieldlines
Y	Adaptive Signed	Y channel of all fieldlines
Z	Adaptive Signed	Z channel of all fieldlines

Table 2: Body of the FITS File

1.1.1 Splitting up the body

After decoding the body the channels X, Y and Z data have to be split up to their corresponding fieldlines. The decoded LENGTH describes the number of data points which belong to the first fieldline. An example is is illustrated in table 3.

decoded LENGTH	4	3	18	...

	first fieldline				second fieldline			...
decoded X	x_0	x_1	x_2	x_3	x_4	x_5	x_6	...
decoded Y	y_0	y_1	y_2	y_3	y_4	y_5	y_6	...
decoded Z	z_0	z_1	z_2	z_3	z_4	z_5	z_6	...

Table 3: Illustration of how the body has to be split up

1.2 Byte Encoding

The Byte encoding which is in use for adapts to the number of Bits needed to represent a value. Hence the name ”Adaptive Precision Encoding”. Two implementations are in use: the signed and unsigned variant. The ”Unsigned Adaptive Precision Encoding” is illustrated in table 4 and the signed version in table 5.

The first Bit of the ”Unsigned Adaptive Precision Encoding” is the ”Continue Flag”. If the ”Continue Flag” is set, then the next Byte also belongs to the value. If the ”Continue Flag” is not set then the next Byte belongs to a new value. The Bytes are written in Big-Endian[3] convention. Which means that the first data Bit is the Most-Significant-Bit.

The ”Signed Adaptive Precision Encoding” only has one difference when compared to the unsigned encoding: The first Byte of a value carries the ”Sign Flag” as shown in table 5. If the following Byte belongs to the same value then the encoding of the second Byte is identical to the unsigned encoding (see table 4).

Unsigned Adaptive Precision Encoding

Byte							
Continue Flag	X	X	X	X	X	X	X

Table 4: Unsigned Adaptive Byte Encoding. X are data Bits.

Signed Adaptive Precision Encoding

Byte							
Continue Flag	Sign Flag	X	X	X	X	X	X

Table 5: Signed Adaptive Byte Encoding of the First Byte. X are data Bits.

Here is an example implementation of the ”Signed Adaptive Precision Encoding”:

```
public static final int continueFlag = 128;
public static final int signFlag = 64;
public static final int dataBitCount = 7;

public static int[] decodeAdaptive(byte[] data) {
  int length = calcLength(data);
  int[] output = new int[length];
  int outIndex = 0;

  //for each encoded byte
```

```
    for (int i = 0; i < data.length; i++) {
      byte current = data[i];
      int value = (short) (current & (signFlag - 1));
      int minus = -(current & signFlag);

      //add encoded bytes as long as the continue flag is set.
      boolean run = (current & continueFlag) != 0;
      while (run) {
        current = data[++i];
        run = (current & continueFlag) != 0;
        minus <<= dataBitCount;
        value <<= dataBitCount;
        value += current & (continueFlag - 1);
      }
      output[outIndex++] = (value + minus);
    }

    return output;
  }

  private static int calcLength(byte[] data) {
    int out = 0;

    for (int i = 0; i < data.length; i++) {
      if ((data[i] & continueFlag) == 0)
        out++;
    }
    return out;
  }
```

1.3 Decode Prediction

Here a recursive algorithm is used to predict the content of a channel. The prediction assumes that the mid-value of the channel is on a straight line between the start and end value. Only the error of the prediciton is saved. The figure 1 illustrates this. After the first prediction the algorithm is repeated recursively for the two new segment. The figure illustrates 2 the second recursive step. The recursion stops when all values in a channel have been predicted.

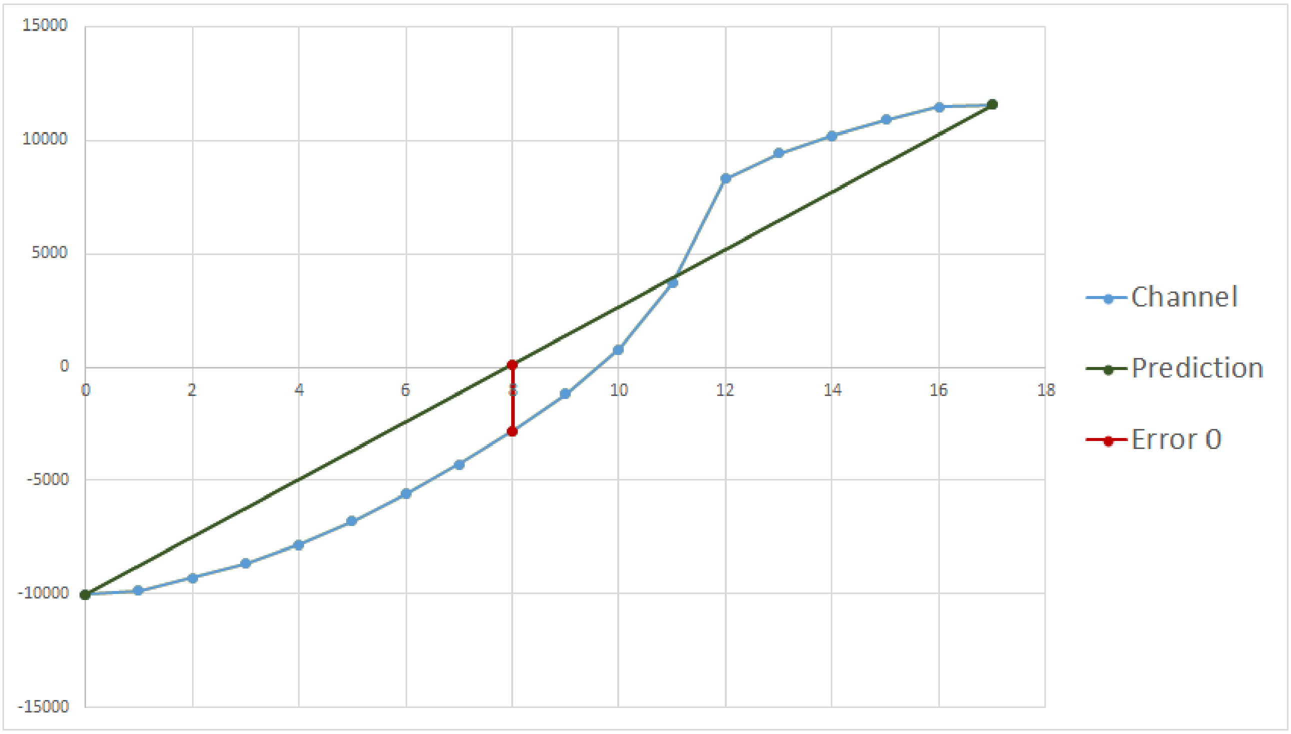

Figure 1: First recursive step.

After Byte decoding and splitting, a channel contains the start and end value and the prediction error as illustrated in table 6.

Channel					
start value	endvalue	Error 0	Error 1	Error 2	...

Table 6: Content of a channel

The content of the channel is quantized. The content has to be multiplied with the quantization factors Q_1,Q_2 and Q_3 from the FITS file header (see table 1):

$$\begin{aligned} &\text{multiply with } Q_1 \text{ if index } < 5 \\ &\text{multiply with } Q_2 \text{ if } 5 \leq \text{ index } < 16 \\ &\text{multiply with } Q_3 \text{ if } 16 \leq \text{ index} \end{aligned} \tag{1.1}$$

Now the channel can be reconstructed:

1. Predict that the middle value of the channel is on a straight line between the start and end value.
2. Substract the prediction error.

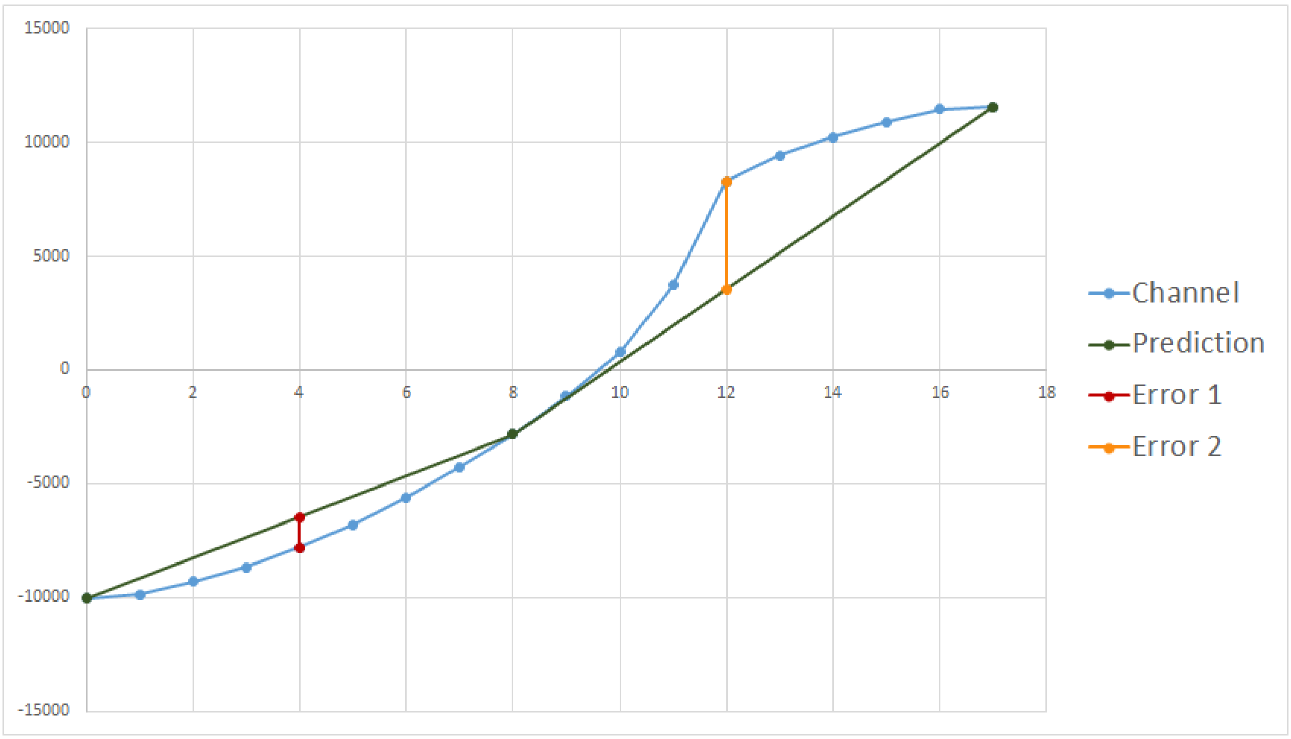

Figure 2: Second recursive step.

3. Repeat for the two new segments

Here is an example implementation in Java:

```java
private float[][] channels;
private int size;

public void decodePrediction(float _Q1, float _Q2, float _Q3)
{
  dequantizePredictionErrors(_Q1,_Q2,_Q3);

  for(int i = 0; i < channels.length;i++) {
    float[] decodedChannel = new float[channels[i].length];
    decodedChannel[0] = channels[i][0];
    decodedChannel[decodedChannel.length-1] = channels[i][0]+channels[i][1];
    if(decodedChannel.length > 2) {
      int channelIndex = 2;
      LinkedList<Indices> queue = new LinkedList<>();
      queue.add(new Indices(0, decodedChannel.length-1));
      while(!queue.isEmpty()) {
        prediction(queue,decodedChannel,channels[i],channelIndex);
        channelIndex++;
      }
    }
    this.channels[i] = decodedChannel;
  }
  this.size = this.channels[0].length;
}

private void dequantizePredictionErrors(float _Q1, float _Q2, float _Q3) {
  for(int i = 0; i < channels.length;i++) {
    float[] current = channels[i];

    int j=0;
    for(; j < 5 && j< current.length;j++) {
      current[j] *= _Q1;
    }
    for(; j < 16 && j< current.length;j++) {
      current[j] *= _Q2;
    }

    for(;  j < current.length;j++) {
      current[j] *= _Q3;
    }
  }
}

private static void prediction(LinkedList<Indices> queue,float[] decodedChannel,
    float[] encodedChanel, int nextIndex) {
  Indices i = queue.pollFirst();
  float start = decodedChannel[i.startIndex];
  float end = decodedChannel[i.endIndex];

  int toPredictIndex = (i.startIndex + i.endIndex) / 2;
    float predictionError = encodedChanel[nextIndex];

    //predict
  float predictionFactor = (toPredictIndex-i.startIndex)/(float)(i.endIndex - i.
      startIndex);
  float prediction = (1-predictionFactor)* start + predictionFactor*end;
  decodedChannel[toPredictIndex] = prediction-predictionError;

  //add next level of indices
  if (i.startIndex + 1 != toPredictIndex){
    Indices next = new Indices(i.startIndex,toPredictIndex);
    queue.addLast(next);
        }
  if (i.endIndex - 1 != toPredictIndex) {
    Indices next = new Indices(toPredictIndex,i.endIndex);
    queue.addLast(next);
  }
}

private static class Indices {
  public int startIndex;
  public int endIndex;

  public Indices(int start, int end) {
    this.startIndex = start;
    this.endIndex = end;
  }
}
```

References

[1] NASA. The FITS support office, November 2014. [Online; accessed 18-November-2014].

[2] Alexander Roshal. rarlab, November 2014. [Online; accessed 18-November-2014].

[3] Wikipedia. Endianness — Wikipedia, the free encyclopedia, 2014. [Online; accessed 30-November-2014].