# Partition Run Length: ImagePlus File Format for Partition Storage

Jordi Pont Tuset and Ferran Marques Acosta

## I. Introduction

In the ImagePlus framework, a partition of an image (`ImagePartition`) is understood as a regular grayscale image whose pixel values are interpreted as labels, i.e., an integer value that identifies the region containing that particular pixel.

As an image, the first attempt to save them to disk was to use any usual lossless format such as BMP, PNG, etc. The common versions of these formats, however, use 8 bits per channel, allowing the partitions to have at most 256 regions. This limitation is clearly not affordable, since many applications require many more regions in the partitions.

It was taken the decision that the default type of data used to represent the region labels was unsigned integers of 32 bits (`uint32`). Although the BMP and PNG standards support this pixel depth, the library used in ImagePlus to read and write images (`ImageMagick`) does not support it.

In order to solve this issue, it was decided to implement a new format based on the run-length technique, which was called Partition Run Length (PRL). Following, we describe this format, including the file header, the coding scheme, and an example.

## II. File format

The Partition Run Length format is a particular case of the MultiArray format used in ImagePlus, so they share the header structure, with a particular value in the `ct` (*Compression Type*) field. The general extension for the MultiArray format is `.mult`. In order to distinguish the PRL format, however, the extension `.prl` may be used instead.

Below, we present the file header format and the coding scheme.

### A. File header

The file header for the PRL format is presented in the following table, where all the numbers are coded as unsigned integers. The 6 first fields (above the line) are common to all the MultiArray files, and the last one (`nb`) is just used in the PRL case.

| Size (bytes) | | Field | Value in PRL |
|---|---|---|---|
| 2 | (`uint16`) | `mn` | 255 |
| 1 | (`uint8`) | `ft` | 1 |
| 1 | (`uint8`) | `ct` | 1 |
| 1 | (`uint8`) | `dt` | 4 |
| 8 | (`uint64`) | `nd` | 2 |
| `nd`·8 | (`nd`·`uint64`) | `sd` | - |
| 1 | (`uint8`) | `nb` | - |

In the case of PRL, therefore, the size of the header is 30 bytes. Following, we give a brief description of all the fields:

- `mn`: *Magic Number*. It allows us to check that the reading and the writing was done in the same little/big-endian machine. It is usually known as Byte Order Mark (BOM).
- `ft`: *File Type*. Type of file being written. In the case of MultiArray, its value is 1.
- `ct`: *Compression Type*. Algorithm used to compress the data. It can be `BYTE_RUNLENGTH` (0) if a general runlength at byte level is used. In the case of PRL, the value of this parameter is `PARTITION_RUNLENGTH` (1).
- `dt`: *Data Type*. Type of data of the MultiArray (`INT8`, `UINT32`, `FLOAT64`, etc.). In the case of PRL, the partitions are stored as `UINT32` (4).
- `nd`: *Number of Dimensions* of the MultiArray. In the case of ImagePartition, this value is 2.
- `sd`: *Size of Each Dimension*.
- `nb`: *Number of Bits* used in some parts of the PRL coding process. It is computed as follows when coding a partition:

$$\texttt{nb} = \left\lceil \log_2(\max + 1) \right\rceil$$

where max is the maximum label of the partition. When decoding a file, this value is simply read from the file.

### B. Partition Run Length

The PRL format is based on a run-length transformation of the partition and a specific coding scheme for both the lengths and the labels. The partition is read from left to right and up to down and transformed into pairs (`label`, `length`) according to the run-length algorithm. In other words, `length` is the number of repetitions of `label` according to the scanning order presented.

These pairs are written consecutively in the file (in binary). The size of `length` is limited to 255, so if the actual length is greater, it is separated into two or more runs.

Following, we describe the way in which both values are coded.

*1) `label` coding:* The binary coded values for the labels are presented in the following table:

| Code | Meaning |
|---|---|
| `0` | Copy the label of the pixel just above the current one. |
| `10` | Copy the label of the following run of the pixel just above the current one. |
| `110` | The current maximum label appeared + 1. |
| `111` | None of the above holds. After the code, `nb` bits follow coding the value of the label directly as an unsigned integer. |

*2) `length` coding:* The binary coded values for the lengths are presented in the following table:

| Code | Meaning |
|---|---|
| `000` | `length`=1 |
| `001` | `length`=2 |
| `010` | `length`=3 |
| `011` | `length`=4 |
| `1` | `length`>4. After the code, 8 bits follow that code the value of the length directly as an unsigned integer. (Recall that lengths are limited to 255.) |

Note that both codes presented are prefix codes, so they are instantaneously decodable.

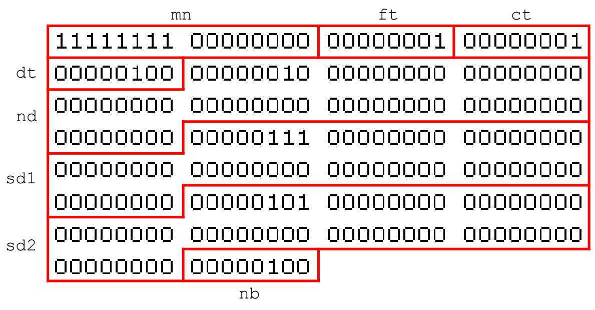
## III. Example

This section is devoted to presenting an example of a PRL coded partition that is simple and small enough for the process to be reproducible manually.

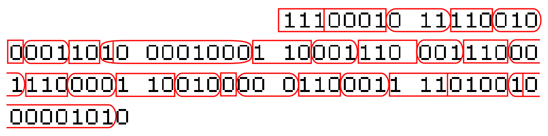
Following, we show the original partition, with each pixel represented by the label of the region it belongs to.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 2 | 2 | 2 |
| 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 3 | 3 | 4 | 4 |
| 5 | 5 | 6 | 7 | 7 | 7 | 4 |
| 8 | 8 | 4 | 4 | 4 | 4 | 4 |

And finally, the header (with each field labeled) and the file content, with the chunks of bits selected referring to the coded labels (squared) and lengths (rounded).



```
11111111 00000000 00000001 00000001
00000100 00000010 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000111 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000101 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000100
```



```
                           11100010 11110010
00011010 00010001 10001110 00111000
11100001 10010000 01100011 11010010
00001010
```