espadon

easy study of patient dicom data in oncology

Raw DICOM handling

C. Fontbonne and J.-M. Fontbonne | LPC Caen
Normandie Univ, ENSICAEN, UNICAEN, CNRS/IN2P3, LPC Caen, 14000 Caen, France
Contact.espadon@lpccaen.in2p3.fr
https://espadon.cnrs.fr/

Table of content

1 Introduction. What we will learn?

DICOM access is extremely simplified in espadon, but for at least two reasons, you may want to access DICOM raw data, namely:

- In order to get information you need, that are not available in the Patient Oriented View (POV), for instance the magnetic field in MRs,
- In order to change the DICOM content, for instance, to check if you favorite dosimeter is able to detect accelerator/multi leaves collimator (MLC) malfunctions.

Changing the DICOM content is extremely dangerous and shall never be applied to patient treatment. We decline any responsible for losses, damages, or injury incurred by your use of these functionalities.

The first section will be devoted to retrieving DICOM information, the second section to DICOM content modifying.

2 Retrieving DICOM information

The first step consists in loading the DICOM file into memory:

```
require (espadon)

dcm.filename <- file.choose ()
dcm <- dicom.raw.data.loader (dcm.filename)
```

Doing this just load “raw” data you cannot exploit…

```
str (dcm)

 raw [1:530278] 00 00 00 00 ...
```

Note: if you use the Rdcm file format, you have access to DICOM content using the `load.Rdcm.raw.data` instruction. It returns a list, and the DICOM data are stored in the `$data` element of this list as a named list of tags whose content is the decoded value of each tag. See §II.4 for handling such object.

2.1 Basics of DICOM storage

The most important instruction is `dicom.parser`. It provides you with a view of the parsed DICOM content stored in an intelligible data frame:

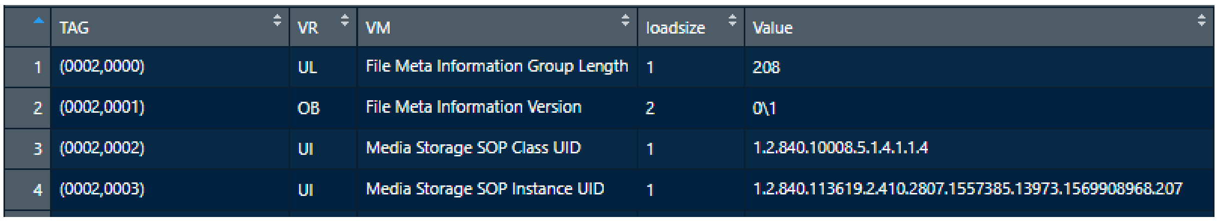
```
L <- dicom.parser (dcm)
str (L)

'data.frame':   380 obs. of  5 variables:
 $ TAG     : chr  "(0002,0000)" "(0002,0001)" "(0002,0002)" "(0002,0003)" ...
 $ VR      : chr  "UL" "OB" "UI" "UI" ...
 $ VM      : chr  "File Meta Information Group Length" "File Meta Information Version" "Media Storage SOP
Class UID" "Media Storage SOP Instance UID" ...
 $ loadsize: int  1 2 1 1 1 1 1 1 1 1 ...
 $ Value   : chr  "208" "0\\1" "1.2.840.10008.5.1.4.1.1.4"
"1.2.840.113619.2.410.2807.1557385.13973.1569908968.207" ...
```

The resulting data frame contains:

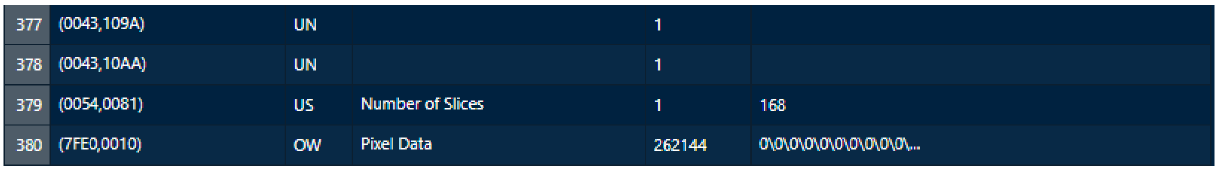
- `TAG`: specific DICOM tags formed by 2 hexadecimal 16 bits words (in upper case) separated by a coma and surrounded by parenthesis, possibly followed by the word 'item' completed by a number.
- `VR`: the value representation as expressed in DICOM format
- `VM`: the value meaning in the DICOM dictionary integrated in espadon
- `loadsize`: the number of objects stored in the tag
- `Value`: the text value of the content, when possible

Executing `View(L)`, you would get something like that (using RStudio):



	TAG	VR	VM	loadsize	Value
1	(0002,0000)	UL	File Meta Information Group Length	1	208
2	(0002,0001)	OB	File Meta Information Version	2	0\1
3	(0002,0002)	UI	Media Storage SOP Class UID	1	1.2.840.10008.5.1.4.1.1.4
4	(0002,0003)	UI	Media Storage SOP Instance UID	1	1.2.840.113619.2.410.2807.1557385.13973.1569908968.207

…



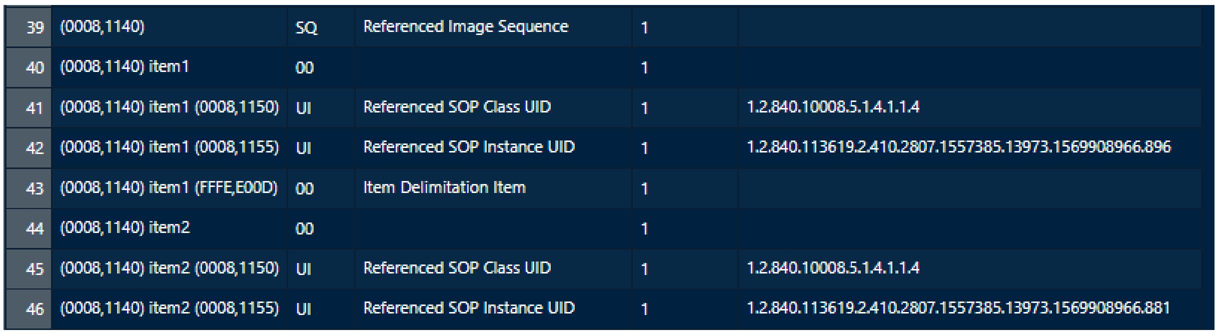
377	(0043,109A)	UN		1	
378	(0043,10AA)	UN		1	
379	(0054,0081)	US	Number of Slices	1	168
380	(7FE0,0010)	OW	Pixel Data	262144	0\0\0\0\0\0\0\0\0\0\...

When `VR` tells us that the tag content is text (`VR` = "AE", "AS", "CS", "DA", "DS", "IS", "LO", "LT", "PN", "SH", "ST", "TM", "UI", "UN", "UT"), the `Value` field exactly represents the text stored (whatever the text string length).

For binary stored tags (`VR` = "OB", "OW", …), see for instance (0002,0001) or (7FE0,0010), the text string is the concatenation of the tag content, limited to `txt.length` characters. When the string is truncated, the it is terminated by "…". Beware that, in espadon the separator used by default for binary stored tags is "\\". It appears explicitly in the text string. However, when represented in RStudio using `View`, only one backslash is visible (but there are actually two).

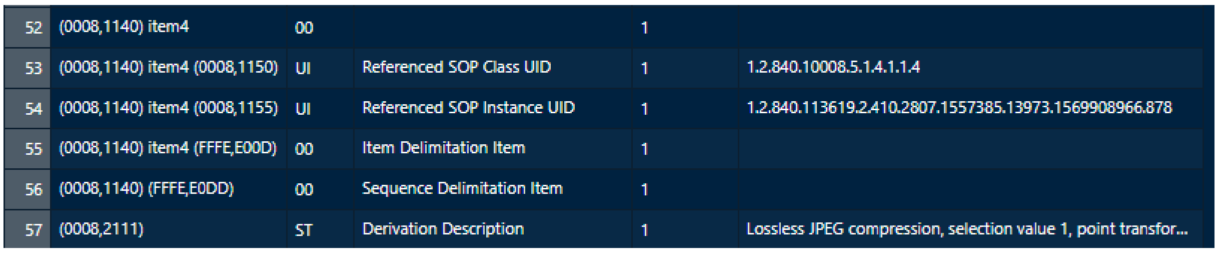
Some tags are assigned "UN" `VR`. This means that this tag is unknown in the DICOM dictionary. It is probably a manufacturer's proprietary tag. In our case, all "UN" tags were anonymized (conservative position, as we do not know what they could contain) and the decoded string is empty. For an original DICOM file, it would not be empty and the content is supposed be in ascii format.

DICOM is in fact an encapsulated format. This aspect can be seen in the example below:



39	(0008,1140)	SQ	Referenced Image Sequence	1	
40	(0008,1140) item1	00		1	
41	(0008,1140) item1 (0008,1150)	UI	Referenced SOP Class UID	1	1.2.840.10008.5.1.4.1.1.4
42	(0008,1140) item1 (0008,1155)	UI	Referenced SOP Instance UID	1	1.2.840.113619.2.410.2807.1557385.13973.1569908966.896
43	(0008,1140) item1 (FFFE,E00D)	00	Item Delimitation Item	1	
44	(0008,1140) item2	00		1	
45	(0008,1140) item2 (0008,1150)	UI	Referenced SOP Class UID	1	1.2.840.10008.5.1.4.1.1.4
46	(0008,1140) item2 (0008,1155)	UI	Referenced SOP Instance UID	1	1.2.840.113619.2.410.2807.1557385.13973.1569908966.881

…



52	(0008,1140) item4	00		1	
53	(0008,1140) item4 (0008,1150)	UI	Referenced SOP Class UID	1	1.2.840.10008.5.1.4.1.1.4
54	(0008,1140) item4 (0008,1155)	UI	Referenced SOP Instance UID	1	1.2.840.113619.2.410.2807.1557385.13973.1569908966.878
55	(0008,1140) item4 (FFFE,E00D)	00	Item Delimitation Item	1	
56	(0008,1140) (FFFE,E0DD)	00	Sequence Delimitation Item	1	
57	(0008,2111)	ST	Derivation Description	1	Lossless JPEG compression, selection value 1, point transfor...

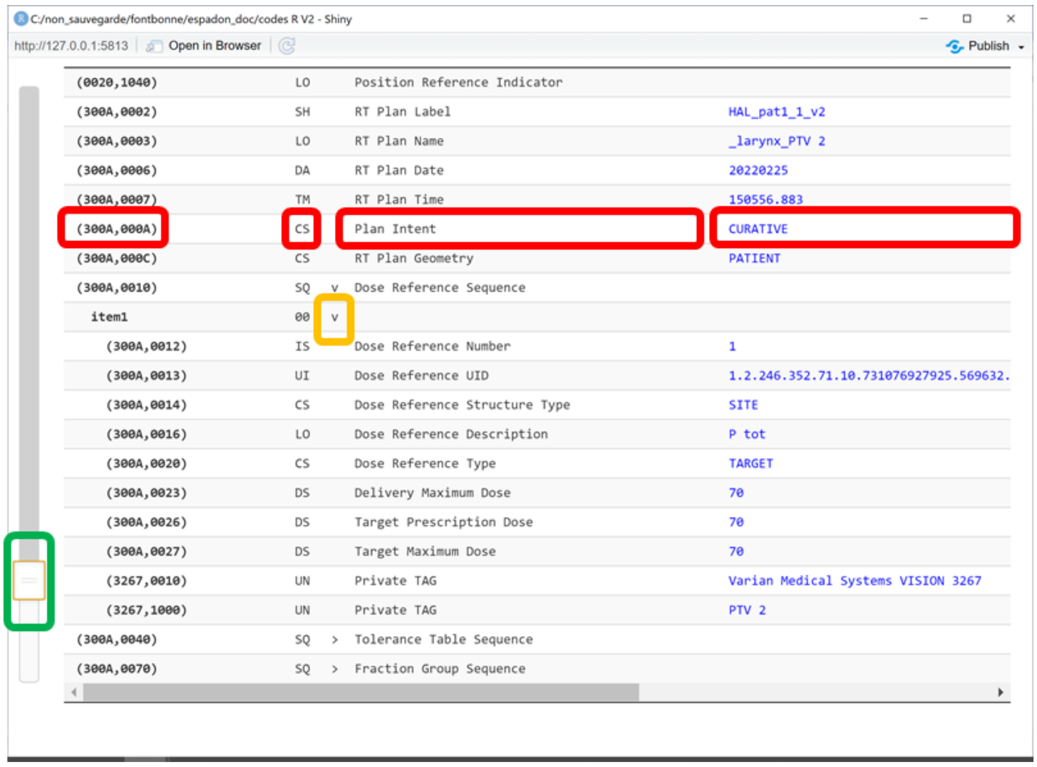
The (0008,1140) tag starts the "Referenced Image Sequence". In this sequence, every tag will begin with (0008,1140). There may be several items in this sequence (here, in instance, 4). The tag is followed by the "item" term, followed by the item number itself (1..4). Then the tag is completed by the tag of the item of the sequence, for instance (0008,1150) forming the final tag, for instance "(0008,1140) item3 (0008,1150)". This tag should be understood as "Referenced SOP Class UID" of the third item in the "Referenced Image Sequence".

2.2 Exploring DICOM tree

Exploring DICOM files content can be a complex task as DICOM formats ensures you have all relevant information, but sometimes stored a strange way. espadon contains a user interface that allows you to navigate at will to examine the architecture of the file and its contents. To do this, simply call `dicom.viewer` function:

```
require (espadon)

dcm.filename <- file.choose ()
dcm <- dicom.raw.data.loader (dcm.filename)
dicom.viewer(dcm)
```



```
(0020,1040)      LO     Position Reference Indicator
(300A,0002)      SH     RT Plan Label                   HAL_pat1_1_v2
(300A,0003)      LO     RT Plan Name                    _larynx_PTV 2
(300A,0006)      DA     RT Plan Date                    20220225
(300A,0007)      TM     RT Plan Time                    150556.883
(300A,000A)      CS     Plan Intent                     CURATIVE
(300A,000C)      CS     RT Plan Geometry                PATIENT
(300A,0010)      SQ  v  Dose Reference Sequence
  item1          00  v
    (300A,0012)  IS     Dose Reference Number           1
    (300A,0013)  UI     Dose Reference UID              1.2.246.352.71.10.731076927925.569632.
    (300A,0014)  CS     Dose Reference Structure Type   SITE
    (300A,0016)  LO     Dose Reference Description      P tot
    (300A,0020)  CS     Dose Reference Type             TARGET
    (300A,0023)  DS     Delivery Maximum Dose           70
    (300A,0026)  DS     Target Prescription Dose        70
    (300A,0027)  DS     Target Maximum Dose             70
    (3267,0010)  UN     Private TAG                     Varian Medical Systems VISION 3267
    (3267,1000)  UN     Private TAG                     PTV 2
(300A,0040)      SQ  >  Tolerance Table Sequence
(300A,0070)      SQ  >  Fraction Group Sequence
```

Espadon DICOM viewer. In red, from left to right: the DICOM TAG or item, its value representation, its meaning and its contain (limited by the number of characters you requested). In green: in order to navigate, use this slider. In orange: if you want to open or close a sequence or an item, click on the arrow.

2.3 Retrieving data

Once the format understood, retrieving data is “almost” easy using the `grepl` instruction.

If you do not know the tags you are looking for, you can search the `VM` field for specific words, for instance:

```
idx <- grepl ("coil", tolower (L$VM), fixed = TRUE)
L[idx, c("TAG", "VR", "VM", "Value")]

            TAG VR                     VM  Value
108 (0018,1250) SH Receive Coil Name Head     24

idx <- grepl ("magnetic", tolower (L$VM), fixed = TRUE)
L[idx, c("TAG", "VR", "VM", "Value")]

           TAG VR                      VM  Value
94 (0018,0087) DS Magnetic Field Strength    1.5

idx <- grepl ("angle", tolower (L$VM), fixed = TRUE)
L[idx, c("TAG", "VR", "VM", "Value")]

            TAG VR                       VM Value
111 (0018,1314) DS               Flip Angle    15
112 (0018,1315) CS Variable Flip Angle Flag     N
```

Beware that such a search can return several results (like “angle” for instance).

Note the `fixed = TRUE` option when calling `grepl`, meaning the pattern (“coil” for instance) is a sentence that should be search exactly as it is written.

Note also that we converted `VM` to lower case to match the pattern. We could have used `ignore.case = TRUE` option when calling `grepl` instead.

If you know the explicit tag, you have several options depending on your needs. Let us illustrate some, by the use of the powerful (but sometimes confusing) `grepl` instruction:

```
idx <- grepl ("(0008,1140)", L$TAG, fixed = TRUE)
L[idx, c("TAG", "VR", "VM", "Value")]


                             TAG VR                          VM                                                 Value
39                   (0008,1140) SQ   Referenced Image Sequence                                                      
40             (0008,1140) item1 00                                                                                  
41 (0008,1140) item1 (0008,1150) UI     Referenced SOP Class UID                            1.2.840.10008.5.1.4.1.1.4
42 (0008,1140) item1 (0008,1155) UI Referenced SOP Instance UID 1.2.840.113619.2.410.2807.1557385.13973.1569908966.896
43 (0008,1140) item1 (FFFE,E00D) 00      Item Delimitation Item                                                      
44             (0008,1140) item2 00                                                                                  
45 (0008,1140) item2 (0008,1150) UI     Referenced SOP Class UID                            1.2.840.10008.5.1.4.1.1.4
46 (0008,1140) item2 (0008,1155) UI Referenced SOP Instance UID 1.2.840.113619.2.410.2807.1557385.13973.1569908966.881
47 (0008,1140) item2 (FFFE,E00D) 00      Item Delimitation Item                                                      
48             (0008,1140) item3 00                                                                                  
49 (0008,1140) item3 (0008,1150) UI     Referenced SOP Class UID                            1.2.840.10008.5.1.4.1.1.4
50 (0008,1140) item3 (0008,1155) UI Referenced SOP Instance UID 1.2.840.113619.2.410.2807.1557385.13973.1569908966.887
51 (0008,1140) item3 (FFFE,E00D) 00      Item Delimitation Item                                                      
52             (0008,1140) item4 00                                                                                  
53 (0008,1140) item4 (0008,1150) UI     Referenced SOP Class UID                            1.2.840.10008.5.1.4.1.1.4
54 (0008,1140) item4 (0008,1155) UI Referenced SOP Instance UID 1.2.840.113619.2.410.2807.1557385.13973.1569908966.878
55 (0008,1140) item4 (FFFE,E00D) 00      Item Delimitation Item
```

As `fixed = TRUE`, `grepl` explicitly searches for the specified tag. We see that (0008,1140) starts a sequence (its VR is “SQ”). This sequence has 4 items, each item having several tags.

`grepl` returns every occurrence of the tag, even if it is not at first position. For instance:

```
idx <- grepl ("(0008,1155)", L$TAG, fixed = TRUE)
L[idx, c("TAG", "VR", "VM", "Value")]


                             TAG VR                          VM                                                 Value
36 (0008,1120) item1 (0008,1155) UI Referenced SOP Instance UID      1.2.124.113532.80.22144.5.20190729.143342.1107833
42 (0008,1140) item1 (0008,1155) UI Referenced SOP Instance UID 1.2.840.113619.2.410.2807.1557385.13973.1569908966.896
46 (0008,1140) item2 (0008,1155) UI Referenced SOP Instance UID 1.2.840.113619.2.410.2807.1557385.13973.1569908966.881
50 (0008,1140) item3 (0008,1155) UI Referenced SOP Instance UID 1.2.840.113619.2.410.2807.1557385.13973.1569908966.887
54 (0008,1140) item4 (0008,1155) UI Referenced SOP Instance UID 1.2.840.113619.2.410.2807.1557385.13973.1569908966.878
```

Here we get the lines previously discovered (42 to 54) searching for (0008,1140), and we learn that this tag is also used in (0008,1120). One thing important to remind on tags is that a tag is just a meaning. So far, it can be used in any sequence, if its meaning is correct in the context. Thus, searching for explicit tags can be hazardous…

Last, it is important to note that `grepl` uses regular expressions (regexp) that are not text (unless `fixed = TRUE`), but more or less complex character sequences we want to look for. For instance, we learned that `(0008,1140)` starts a sequence. A good question could be: “how many items are there in this sequence”? To answer this question, try this:

```
idx <- grepl ("^[(]0008,1140[)] item[[:digit:]]+$", L$TAG)
L[idx, c("TAG", "VR", "VM", "Value")]

                  TAG VR VM Value
40 (0008,1140) item1 00
44 (0008,1140) item2 00
48 (0008,1140) item3 00
52 (0008,1140) item4 00
```

Here, we truly used a regexp, and clearly, it need some explanations:

1) We are searching for “0008,1140” string.
2) This string should be in parenthesis. Parenthesis have a specific meaning in regexp, if we would have begun our regexp by “`(0008,1140)`”, we would have got… nothing. In order to tell the regexp we explicitly want parenthesis, we must enclose them in square brackets: “`[(]`” and “`[)]`”.
3) We want “`(0008,1140)`” be exactly at the beginning of the sentence. This is the role of “^”.
4) Next, we want “ item”,
5) Followed by a number. That is why “item” is followed by “`[[:digit:]]`”, meaning we want any number in 0-9.
6) As we do not know the number (it could exceed 9, as for instance “10” or more), we tell the regexp the digit could be repeated. This is the “`+`”.
7) Last, we want this digit conclude the string, that is the “`$`”.

In other words, we are searching for something “exactly” like that: “(0008,1140) itemX..X” where “X” is a digit. Well, regexp are powerful, but exhausting… Yet, we got the item delimiters of the sequence, and we can search further into each item.

As we know the number of items, we can now search for data of the third item (for instance):

```
idx <- grepl ("^[(]0008,1140[)] item3", L$TAG)
L[idx, c("TAG", "VR", "VM", "Value")]

                             TAG VR                          VM                                                  Value
48             (0008,1140) item3 00
49 (0008,1140) item3 (0008,1150) UI    Referenced SOP Class UID                              1.2.840.10008.5.1.4.1.1.4
50 (0008,1140) item3 (0008,1155) UI Referenced SOP Instance UID 1.2.840.113619.2.410.2807.1557385.13973.1569908966.887
51 (0008,1140) item3 (FFFE,E00D) 00      Item Delimitation Item
```

Beware that this search would have given you a positive response for “(0008,1140) item 30” also… Could you find, knowing the tag structure, how to exclude (or include) numbers with several digits?

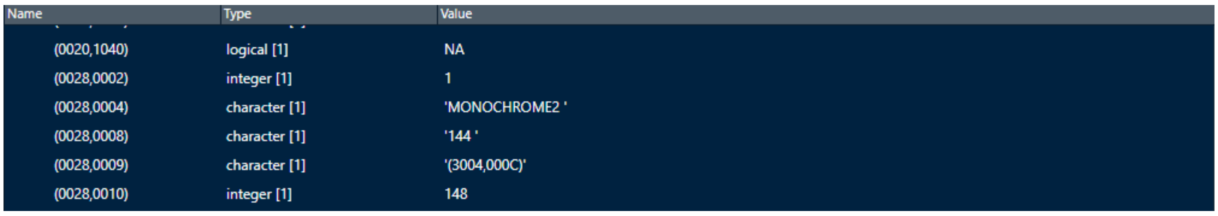
Answer: just add an empty space at the end of the sentence.

2.4 List representation

For the moment, we just accessed character strings stored in a data frame. It is usually sufficient as espadon incorporates mechanisms to handle relevant information and images for DICOM in oncology. In the case of other modalities, it can be useful to access raw data. If you are in this case, consider using the `dicom.parser` instruction with `as.txt = FALSE` option. It will give you a list of DICOM tags decoded according DICOM format:

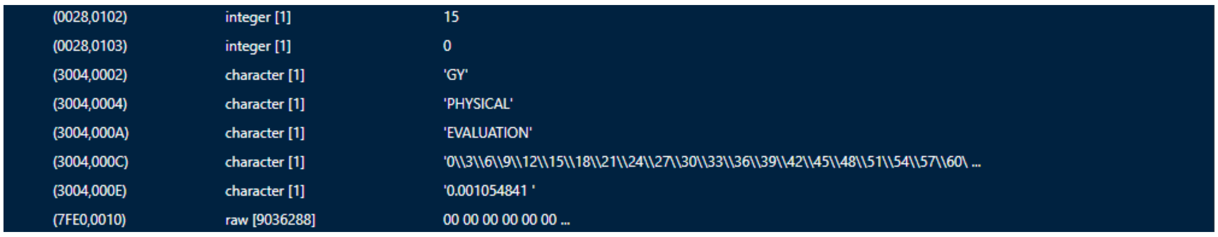
```
L <- dicom.parser (dcm, as.txt = FALSE)
str (L)

List of 54
 $ (0002,0000): num 200
 $ (0002,0001): int [1:2] 0 1
 $ (0002,0002): chr "1.2.840.10008.5.1.4.1.1.481.2"
 $ (0002,0003): chr "1.2.752.243.1.1.20200625182835514.4000.38473"
```



Name	Type	Value
(0020,1040)	logical [1]	NA
(0028,0002)	integer [1]	1
(0028,0004)	character [1]	'MONOCHROME2 '
(0028,0008)	character [1]	'144 '
(0028,0009)	character [1]	'(3004,000C)'
(0028,0010)	integer [1]	148

…



(0028,0102)	integer [1]	15
(0028,0103)	integer [1]	0
(3004,0002)	character [1]	'GY'
(3004,0004)	character [1]	'PHYSICAL'
(3004,000A)	character [1]	'EVALUATION'
(3004,000C)	character [1]	'0\\3\\6\\9\\12\\15\\18\\21\\24\\27\30\\33\\36\\39\\42\\45\\48\\51\\54\\57\\60\ ...
(3004,000E)	character [1]	'0.001054841 '
(7FE0,0010)	raw [9036288]	00 00 00 00 00 00 ...

This way, every tag is decoded and can be accessed for processing, as for instance (7FE0,0010), which is an image. Obviously, in this case, all the processing of the image is of your concern and you will need to find ways to decode it properly.

Accessing each tag uses the same string strategies described above except, we now have a list, for instance:

```
L$"(0008,0060)"

[1] "RTDOSE"

L[["(0008,0060)"]]

[1] "RTDOSE"
```

2.5 Use of proprietary dictionary

Espadon contains a DICOM tags dictionary you can access with `dicom.tag.dictionary`. This dictionary is by default the merge of DICOM dictionary from nema (https://www.dicomstandard.org/current) and Raysearch (https://www.raysearchlabs.com/4aaf2e/siteassets/raystation-landing-page/dicom-conformance-statements/raystation-pdfs/rsl-d-rs-11a-dcs-en-1.0-2021-05-07-raystation-11a-dicom-conformance-statement.pdf). It is possible to use only the nema dictionary, by assigning the parameter `add.dict` to NULL. The `dicom.tag.dictionary` contains the DICOM translation of tags (upper case), tags content and the Value Representation of the tags content (i.e. encoding mode).

```
str (dicom.tag.dictionary ())

'data.frame':   5925 obs. of  3 variables:
 $ tag : chr  "(0000,0000)" "(0000,0001)" "(0000,0002)" "(0000,0003)" ...
 $ name: chr  "Command Group Length" "Command Length to End" "Affected SOP Class UID" "Requested SOP Class UID" ...
 $ VR  : chr  "UL" "UL" "UI" "UI" ...
```

It is in fact a simple data frame, which is provided, by default to any function requiring it.

You may also have access to proprietary (i.e. manufacturers) dictionaries (which was not our case, as illustrated above). There is a mechanism in espadon to integrate these dictionaries and get appropriate data representations and meaning.

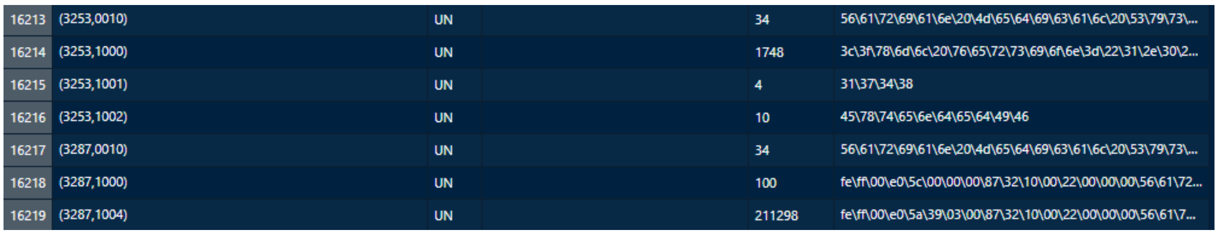
You "just" have to:

1) check your dictionary is strictly conform to espadon dictionary structure as shown above,
2) merge both, using for instance: `my.dicom.tag.dictionary = cbind (dicom.tag.dictionary (), the.dictionary.I.just.loaded)`,
3) check there are no redundancies in tags, for instance, check `length (unique (my.dicom.tag.dictionary$tag)) == length (my.dicom.tag.dictionary$tag)` is TRUE
4) and use the resulting dictionary in instructions like for instance `dicom.parser (dicom.raw.data, …, tag.dictionary = my.dicom.tag.dictionary)`

You should have your new tags decoded well.

If you do not have access to a proprietary dictionary, several solutions exist, for instance:

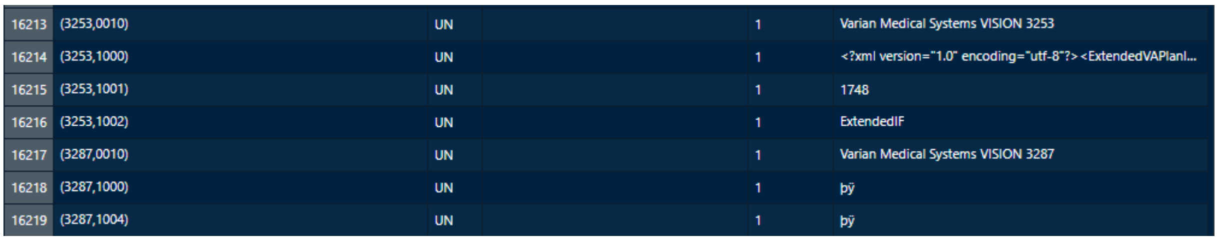
```
L <- dicom.parser (dcm, txt.length = 100)
```



16213	(3253,0010)	UN		34	56\61\72\69\61\6e\20\4d\65\64\69\63\61\6c\20\53\79\73\...
16214	(3253,1000)	UN		1748	3c\3f\78\6d\6c\20\76\65\72\73\69\6f\6e\3d\22\31\2e\30\2...
16215	(3253,1001)	UN		4	31\37\34\38
16216	(3253,1002)	UN		10	45\78\74\65\6e\64\65\64\49\46
16217	(3287,0010)	UN		34	56\61\72\69\61\6e\20\4d\65\64\69\63\61\6c\20\53\79\73\...
16218	(3287,1000)	UN		100	fe\ff\00\e0\5c\00\00\00\87\32\10\00\22\00\00\00\56\61\72...
16219	(3287,1004)	UN		211298	fe\ff\00\e0\5a\39\03\00\87\32\10\00\22\00\00\00\56\61\7...

Here, you have access to the bytes content of "UN" VR fields. If you try:

```
L <- dicom.parser (dcm, try.parse = TRUE, txt.length = 100)
```



16213	(3253,0010)	UN		1	Varian Medical Systems VISION 3253
16214	(3253,1000)	UN		1	<?xml version="1.0" encoding="utf-8"?><ExtendedVAPlanI...
16215	(3253,1001)	UN		1	1748
16216	(3253,1002)	UN		1	ExtendedIF
16217	(3287,0010)	UN		1	Varian Medical Systems VISION 3287
16218	(3287,1000)	UN		1	þÿ
16219	(3287,1004)	UN		1	þÿ

The fields are decoded as text, when possible, even if we do not know their meaning. Some of them become explicit, as `(3287,0010)`, some of them stay unreadable, as `(3287,1000)`. When `try.parse = TRUE`, espadon convert raw data into text, until it finds a "00" byte.

BEWARE that, when using the pseudonymisation functionality of espadon, every field of "UN" VR is emptied because it could have contented free text (patient information for instance).

2.6 Example: reading and displaying a rtimage file

rt-Image belong to the formats not included natively in espadon for reasons already explained (too dependent on manufacturers and devices). If you are interested in using such files, you can try something like that:

Imaging objects often contain important data for image retrieving, as for instance:

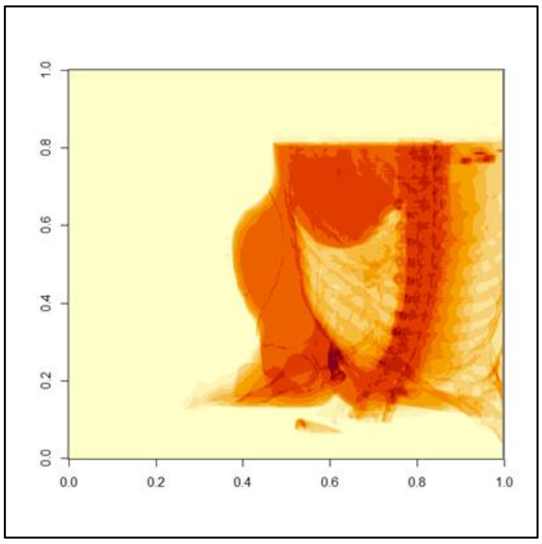
tag	meaning	Value in my example
(0028,0010)	Rows	512
(0028,0011)	Columns	512
(0028,0100)	Bits allocated	16
(0028,0101)	Bits stored	16
(0028,0102)	High Bit	15

And far more in this case, such as gantry angle, jaws... But we know that we want probably read a 512x512 pixels image, stored as unsigned words.

Then, the image is stored in the `(7FE0,0010)` tag. It is exactly 524288 Bytes (512x512x2) as expected.

We could try to read it:

```
dcm.filename <- file.choose ()
dcm <- dicom.raw.data.loader (dcm.filename)
L <- dicom.parser (dcm, as.txt = FALSE)
raw.data <- L[["(7FE0,0010)"]]
raw.image <- readBin(raw.data, what="integer", size = 2, n=length (img) / 2)
my.image <- matrix (raw.image, ncol = 512)
dev.new (width = 7, height = 7, noRStudioGD = T)
image (my.image, useRaster = TRUE)
```



rtimage

Well, it works with 8 lines of code, even if it is only the beginning of the story. Obviously, in order to use such images, you probably would have to read other tags to precisely know the geometry of your image(s) (gantry angle, imaging device position, pixel sizes in mm, intensity meaning, …).

3 Changing DICOM content

We learned, in the previous chapter, how to navigate into DICOM content. Here, we will learn how to change it and save it as a new DICOM file.

Changing DICOM content can be useful for physicists, in order to check their ability to detect delivery faults, but it can be extremely hazardous. DO NOT USE THIS FUNCTIONALITY if you are not precisely sure what you are doing! The example below is given without guaranty of success.

The main instruction to change DICOM content is `dicom.set.tag.value`. **It operates only on text fields** in the DICOM raw data. As espadon does not know produce DICOMs, **you cannot add new fields, just modify existing text fields**. Text fields have a `VR` in {"AE", "AS", "CS", "DA", "DS", "IS", "LO", "LT", "PN", "SH", "ST", "TM", "UI", "UN", "UT"}. `dicom.set.tag.value` is vectored, since it is time consuming, especially for rt-Plan or rt-Stuct. The best way is to prepare the changing collection, and apply it once, just before saving the modified DICOM file.

In this example, we will work on a rt-Plan for a Halcyon machine equipped with a stacked-and-staggered dual-layer multileaf collimator (MLC), consisting of a distal (MLCX1) and a proximal (MLCX2) layer. We want to change a given leaf position in order to simulate a MLC malfunction

The first step consists in understanding the file structure, and then change appropriate fields.

3.1 My rtplan file structure

The `(300A,00B0)` tag contains the "Beam Sequence" stored as items, as the plan can contains several beams. A "beam Sequence" item (may/shall/should, depending on your manufacturer) contains:

- `(300A,00CE)`, the "Treatment Delivery Type". Note it is an optional field. We will only use tag value "TREATMENT" as a beam can also be used for portal imaging, for instance.
- `(300A,0110)`, the "Number of Control Points".
- `(300A,00B6)` which is the description of "Beam Limiting Device Sequence". Each item contains:
  - `(300A,00B8)`, the "RT Beam Limiting Device Type" whose `Value` is in {'X', 'Y', 'ASYMX', 'ASYMY', 'MLCX1', 'MLCX2'}
  - `(300A,00BC)`, the "Number of Leaf/Jaw Pairs"
  - `(300A,00BE)`, the "Leaf Position Boundaries"
- `(300A,0111)` which is the "Control Point Sequence" that interests us. It contains:
  - `(300A,011E)`, the "Gantry Angle". We may change its value for simulating a gantry malfunction, for instance.
  - `(300A,011A)`, the "Beam Limiting Device Position Sequence". It contains:
    - `(300A,00B8)`, the "RT Beam Limiting Device Type" whose `Value` is in {'X', 'Y', 'ASYMX', 'ASYMY', 'MLCX1', 'MLCX2'}, as previously quoted above.
    - `(300A,011C)`, finally, the "Leaf/Jaw Positions". Is is a text vector. *Positions of beam limiting device (collimator) leaf (element) or jaw pairs (in mm) in IEC BEAM LIMITING DEVICE coordinate axis appropriate to RT Beam Limiting Device Type (300A,00B8), e.g., X-axis for MLCX, Y-axis for MLCY. Contains 2N values, where N is the Number of Leaf/Jaw Pairs (300A,00BC) in Beam Limiting Device Sequence (300A,00B6). Values*

shall be listed in IEC leaf (element) subscript order 101, 102, … 1N, 201, 202, … 2N, as quoted in DICOM PS3.3 reference.

For our specific device, executing:

```
dcm.filename <- file.choose ()
dcm <- dicom.raw.data.loader (dcm.filename)

list.to.df <- function (L) {
  last.tag <- sapply(strsplit(names (L),'[ ]'), function (l) rev(l)[1])
  db <- dicom.tag.dictionary()[ match(last.tag, dicom.tag.dictionary()$tag), c("VR","name")]
  db$TAG <-names(L)
  db$Value <- sapply(L, function (l) {
    if (length(l)==0) return("")
    if (is.na(l[1])) return("")
    if (length(l)>1) return(paste("vector of",length(l),"elements"))
    return(as.character(l))
  })
  db <- db[, c(3,1:2,4)]
  colnames (db )<- c("TAG","VR","VM","Value")
  db$VR [grep("^item", last.tag)] <- "00"
  db$VR [is.na(db$VR)] <- "UN"
  db$VM[is.na(db$VM)] <- ""
  return (db)
}
L <- list.to.df (dicom.parser (dcm, as.txt = FALSE))

idx <- grepl ("^[(]300A,00B0[)] item[[:digit:]]+ [(]300A,00B6[)] item[[:digit:]]+ [(]300A,00B8[)]", L$TAG)
unique (L[idx, "Value"])

[1] "X "     "Y "     "MLCX1 " "MLCX2 "
```

tells us we have 2 multi leaf collimators called “MLCX1” and “MLCX2”. Note the spaces in the answer; in DICOM format, strings often contain an even number of characters.

Now, we would like to know how many leaves has each MLC:

```
idx1 <- grepl ("^[(]300A,00B0[)] item[[:digit:]]+ [(]300A,00B6[)] item[[:digit:]]+ [(]300A,00B8[)]", L$TAG)
idx2 <- grepl ("^[(]300A,00B0[)] item[[:digit:]]+ [(]300A,00B6[)] item[[:digit:]]+ [(]300A,00BC[)]", L$TAG)
table (L[idx1, c ("Value")], L[idx2, c ("Value")])


         1  28 29
  MLCX1   0  4  0
  MLCX2   0  0  4
  X       4  0  0
  Y       4  0  0
```

MLCX1 has 28 leaf pairs, and MLCX2, 29.

If we admit (as we checked) that each beam uses the same beam limiting devices, we can get the boundaries for each limiting device, here for instance, MLCX2:

```
idx <- grepl ("^[(]300A,00B0[)] item1 [(]300A,00B6[)] item4", L$TAG)
L[idx, c("TAG", "VR", "VM", "Value")]

                                                TAG VR                              VM
169                (300A,00B0) item1 (300A,00B6) item4 00
170 (300A,00B0) item1 (300A,00B6) item4 (300A,00B8) CS RT Beam Limiting Device Type
171 (300A,00B0) item1 (300A,00B6) item4 (300A,00BC) IS      Number of Leaf/Jaw Pairs
172 (300A,00B0) item1 (300A,00B6) item4 (300A,00BE) DS      Leaf Position Boundaries
173 (300A,00B0) item1 (300A,00B6) item4 (FFFE,E00D) 00        Item Delimitation Item

Value
169
```

```
170
MLCX2
171
29
172 -145\\-135\\-125\\-115\\-105\\-95\\-85\\-75\\-65\\-55\\-45\\-35\\-25\\-15\\-
5\\5\\15\\25\\35\\45\\55\\65\\75\\85\\95\\105\\115\\125\\135\\145
173
```

There are 30 boundaries, as we have 29 leaves.

3.2 Changing leaves potisions

In order to change DICOM content, we will use the `dicom.set.tag.value` instruction. It needs a vector of tags to change and a vector of new tags values.

Suppose we would like to simulate a MLC malfunction affecting MLCX2 where the leaves 13 to 15 of both sides are -10 mm away their actual position. The code could be something like that:

First, we need to find every tag corresponding to MLCX2 and get leaves positions:

```
MLC <- "MLCX2"
idx1 <- grepl (MLC, L$Value)
all.MLCX2.tags <- L$TAG[idx1]
head (all.MLCX2.tags)

[1] "(300A,0040) item1 (300A,0048) item6 (300A,00B8)"
[2] "(300A,00B0) item1 (300A,00B6) item4 (300A,00B8)"
[3] "(300A,00B0) item1 (300A,0111) item1 (300A,011A) item4 (300A,00B8)"
[4] "(300A,00B0) item1 (300A,0111) item2 (300A,011A) item2 (300A,00B8)"
[5] "(300A,00B0) item1 (300A,0111) item3 (300A,011A) item2 (300A,00B8)"
[6] "(300A,00B0) item1 (300A,0111) item4 (300A,011A) item2 (300A,00B8)"
```

This selection gives all references related to MLCX2, not only leaves positions, but also boundary limits,… We must restrict our search to (300A,011A) tag:

```
idx2 <- grepl ("[(]300A,011A[)]", all.MLCX2.tags)
```

In fact, we are not interested in (300A,00B8) tag, but rather in (300A,011C) that contains the leaves positions. We just have to replace the former by the latter:

```
MLC.tags <- strsplit (all.MLCX2.tags[idx2], " ")
MLC.tags <- sapply (MLC.tags, function (tag.encaps) paste (c (rev (rev (tag.encaps)[-1]), "(300A,011C)"),
collapse = " "))
head (MLC.tags)

[1] "(300A,00B0) item1 (300A,0111) item1 (300A,011A) item4 (300A,011C)"
[2] "(300A,00B0) item1 (300A,0111) item2 (300A,011A) item2 (300A,011C)"
[3] "(300A,00B0) item1 (300A,0111) item3 (300A,011A) item2 (300A,011C)"
[4] "(300A,00B0) item1 (300A,0111) item4 (300A,011A) item2 (300A,011C)"
[5] "(300A,00B0) item1 (300A,0111) item5 (300A,011A) item2 (300A,011C)"
[6] "(300A,00B0) item1 (300A,0111) item6 (300A,011A) item2 (300A,011C)"
```

And retrieve the MLC positions:

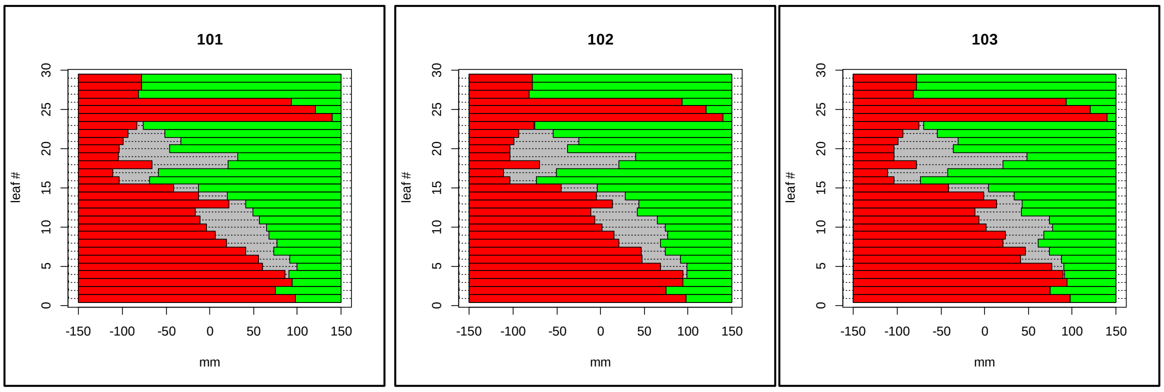
```
my.MLC <- L[L$TAG %in% MLC.tags, ]
```

At this point, it could be a good idea to display some leaves:

```
disp.leaf <- function (leaves, idx) {
  leaf <- as.numeric (strsplit (leaves[idx], "[\\]")[[1]])
  Nl <- length(leaf) / 2
  plot (c(-150, 150), c(1, Nl), typ="n", main = idx, xlab = "mm", ylab = "leaf #")
  rect (-150, .5, 150, Nl+.5, col="grey")
  abline (h=1:Nl, lty=3)
  rect (-150, 1:Nl-0.5, leaf[1:Nl], 1:Nl+0.5, col="red")
  rect (leaf[Nl+1:Nl], 1:Nl-0.5, 150, 1:Nl+0.5, col="green")
}
disp.leaf (my.MLC$Value, 101)
disp.leaf (my.MLC$Value, 102)
disp.leaf (my.MLC$Value, 103)
```

The next step consists in changing leaves positions. In this example, leaves are numbered from 1 to `N.leaves`. Left leaves are in banc 1 and right leaves in banc 2:

```
N.leaves <- 29
leaf <- c(13, 14, 15, 13, 14, 15)
leaf.banc <- c(1, 1, 1, 2, 2, 2)
leaf.delta <- rep (-10, 6)
dummy <- sapply (my.MLC$Value, function (leaves) {
  leaf.pos <- as.numeric (strsplit (leaves, "[\\]")[[1]])
  leaf.pos[(leaf.banc-1) * N.leaves + leaf] <- leaf.pos[(leaf.banc-1) * N.leaves + leaf] + leaf.delta
  return (paste (leaf.pos, collapse="\\"))
})
names (dummy) <- NULL
my.MLC$Value <- dummy
```



Leaves positions MLC X2

<u>Note:</u> it could be a good idea to check for leaves collisions or positions not allowed by the MLC hardware.

Last, we just have to change the raw DICOM and save it with an explicit name:

```
dcm.changed <- dicom.set.tag.value(dcm, my.MLC$TAG, my.MLC$Value)
zz <- file (paste ("new", basename (dcm.filename), sep="_"), "wb")
writeBin(dcm.changed, zz, size = 1)
close (zz)
```

The "new_" file will be saved in your current folder.

If you try to import this file into the TPS that produced the initial rt-Plan, you may get an error message. It is because the rt-Plan is not known only by its filename, but also by its unique identifier, the SOPInstanceUID. So, for obvious security reasons, the TPS do not want to replace the initial rt-Plan by the new one. A way to get around this problem consists in changing the SOPInstanceUID, tag `(0008,0018)`, for a new one, for instance, adding 1 to the last digit.

4 What we learned

Despite everything is made in espadon to avoid the user using and knowing DICOM format, it may be useful in many situation to have a look at the original DICOM content.

In this case, you just have to load your file using the `dicom.raw.data.loader` instruction. Then, depending on your problem, you can access the data using dicom.parser for character string fields as a data frame, or using `dicom.parser (…, as.txt = FALSE)` for decoded or raw data as a tag list.

Then, you will need to find your favorite DICOM fields using, for instance a `grepl` on `L$TAG` in case of a data frame or `names (L)` in case of a list.

If you have a specific DICOM dictionary (coming from a manufacturer), you can incorporate it in espadon.

You can change the content of your DICOM only for character strings `VR`, using the `dicom.set.tag.value` instruction. Finally save this new DICOM.

For more information on DICOM file structure and data representation, have a look at dicomstandard.org website, and begin with a search on PS3.5, for instance.

For more information on DICOM tags, their meaning, their chaining/nesting and far more, have a look at the browser of https://dicom.innolitics.com or execute the code given as example.

Last but not least, don't forget to have a look at espadon manual to discover instruction arguments that could help you.

You can see also :

```
dicom.browser                 DICOM raw data browser
dicom.parser                  Conversion of DICOM raw data into a dataframe or a list of DICOM TAG
information.
dicom.raw.data.anonymizer     DICOM anonymizer
dicom.raw.data.loader         DICOM file loader in raw data
dicom.set.tag.value           Change TAG value in DICOM raw data
dicom.tag.dictionary          DICOM TAG dictionary
dicom.tag.parserDICOM         TAG parser
load.Rdcm.raw.data            Loading a *.Rdcm file.
xlsx.from.dcm                 Converting DICOM files to .xlsx files
```