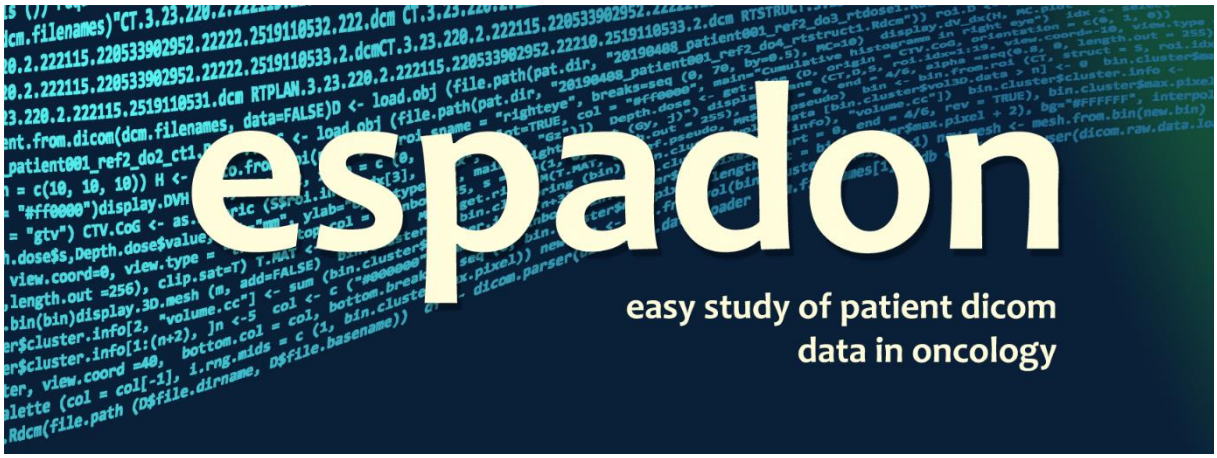



# Automation & efficiency

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?

This document is dedicated to automation & efficiency.

Automation will focus on the methods to be used to carry out studies on large numbers of patients - how to design and test the code - how to sequence it on a multitude of patients - how to manage and detect errors.

Efficiency is another issue. It will rather consist in optimising data transfer times, memory occupation and calculations to be carried out. Indeed, complex studies require frequent revisiting of data and it is always nice to get the job done in minutes rather than hours...

We will illustrate these different methods on a simple example, the calculation of the skin dose. The topic is not important and will be treated in a naive way. Only the methods used here will interest us.

We will start by designing the code on one patient, generalize it to the study of multiple patients, and then see how to optimize it.

# 2 Automation

When we talk about automation, it generally means that we are going to execute one or more codes on a multitude of patients that we assume all have more or less the same data structure. We strongly recommend that you place all these patients in a directory linked to the study, and that you have a directory associated with each patient in this directory.

In the following examples, the study directory is "E:/db" and each patient has his own folder:

```
require (espadon)
require (rgl)
require (png)

pat.db <- "E:/db"
pat.list <- dir (pat.db)
pat.list
```

```
[1] "1"   "10"  "11"  "12"  "13"  "14"  "15"  "16"  "17"  "18"  "18_" "19"
[13] "2"   "20"  "3"   "4"   "6"   "7"   "8"   "9"
```

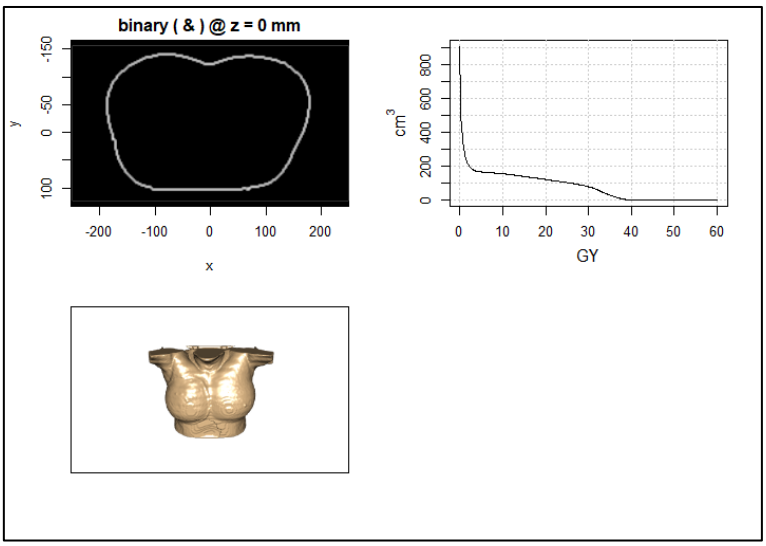
## 2.1 Building the code on a specific patient

To build the code, we start by writing it on a given patient, for example the first one.

We will use the patient contour to make the skin contour, and then we will display the calculated skin contour and the DVH on that contour:

```
require (rgl)
library (png)

pat <- load.patient.from.dicom(file.path (pat.db, pat.list[1]))

S <- load.obj.data (pat$rtstruct[[1]])
D <- load.obj.data (pat$rtdose[[1]])
```

```
B.pat <- bin.from.roi (D, S, roi.name = "body")
B.pat.neg <- bin.inversion (B.pat)
B.pat.neg.expansion <- bin.dilation (B.pat.neg, radius = 3)
B.skin <- bin.intersection (B.pat, B.pat.neg.expansion)

display.plane (B.skin)

D.skin <- vol.from.bin (D, B.skin)
H <- histo.vol (D.skin, breaks = seq (0, 60, by=0.1))
DVH <- histo.DVH (H)

par (mfrow = c (2, 2), mar = c(4,4,2,2))
display.DVH (DVH)

M.pat <- mesh.from.bin (B.pat)

display.3D.mesh (M.pat, col="burlywood")
rgl.snapshot ("dummy.png")
rgl.close ()
png <- readPNG("dummy.png")
plot (c(0, dim (png)[2]+1), c(0, dim (png)[1]+1),
      xlab = "", ylab = "", xaxt="n", yaxt="n",
      asp = 1, xaxs = "i", yaxs = "i", type="n")
rasterImage(png, 1, 1, dim (png)[2], dim (png)[1])
```




execution result of the above code

As we are satisfied with the result, all we have to do is encapsulate the previous code in a function, for example my.skin.dose, which would take the patient's directory as an argument, and run the calculation on all the patients:

```
my.skin.dose <- function (pat.dir) {
  pat <- load.patient.from.dicom(pat.dir)

  S <- load.obj.data (pat$rtstruct[[1]])
  D <- load.obj.data (pat$rtdose[[1]])

  B.pat <- bin.from.roi (D, S, roi.name = "body")
  B.pat.neg <- bin.inversion (B.pat)
  B.pat.neg.expansion <- bin.dilation (B.pat.neg, radius = 3)
  B.skin <- bin.intersection (B.pat, B.pat.neg.expansion)

  par (mfrow = c (2, 2), mar = c(4,4,2,2))
  display.plane (B.skin)

  D.skin <- vol.from.bin (D, B.skin)
```

```
  H <- histo.vol (D.skin, breaks = seq (0, 60, by=0.1))
  DVH <- histo.DVH (H)

  display.DVH (DVH)

  M.pat <- mesh.from.bin (B.pat)

  display.3D.mesh (M.pat, col="burlywood")
  rgl.snapshot ("dummy.png")
  rgl.close ()
  png <- readPNG("dummy.png")
  plot (c(0, dim (png)[2]+1), c(0, dim (png)[1]+1),
      xlab = "", ylab = "", xaxt="n", yaxt="n",
      asp = 1, xaxs = "i", yaxs = "i", type="n")
rasterImage(png, 1, 1, dim (png)[2], dim (png)[1])
}

pdf ("pat skin dose.pdf")
for (p in pat.list) {
  my.skin.dose (file.path (pat.db, p))
}
dev.off ()
```

When you run the code... it crashes after a few patients. Guess why?

Answer: not all patients have a contour called "body", simply.

Of course, in the case illustrated here, it is quite easy to correct, but that is not the point. We will see how to continue the execution despite the error we (deliberately) generated.

## 2.2 Handling execution errors

It is always frustrating to write a code, which can sometimes take several minutes to run on a patient, which has been tested on a few individuals, and which is run on several dozen or even hundreds of cases, and which crashes after several hours of computation, just because a patient does not have the prerequisites for the calculation...

Of course, there is no question of being satisfied with this situation and it is always advisable to understand the reasons for the crash, and to correct them, where possible. However, it is useful to collect what worked, to at least check that it is doing the job properly, at least until you understand what went wrong. To this end, R incorporates error handling mechanisms that allow the code to continue executing by moving on to the next patient, for example. It is up to us later to figure out what happened.

The error handling mechanism uses the `tryCatch` instruction. This instruction tries to execute your function. If everything is Ok, the execution is completed using the `finally` statement. Otherwise, the error message is passed to the `error` statement, which does what it wants with it. The presence of an error does not stop the code which, in the example below, will automatically move on to the next patient:

```
pdf ("pat skin dose.pdf")
Err <- c ()
for (p in pat.list) {
  tryCatch(
    my.skin.dose (file.path (pat.db, p)),
    error = function (e) Err <<- c(Err, p)
    )
}
dev.off ()
```

There are several points to comment on in this example. The first one is that at the end you get the pdf file of the patients for whom the calculation went well. The second is that we would like to have a list of the patients who went wrong. This is the role of the `Err` vector which is incremented by the `error` function which will add the names of the patients for whom the execution has caused problems. As the error handling is itself a function, it is not able to change directly the `Err` vector unless we treat it as a global variable defined outside the statement. This is the role of the "`<<-`" to apply the modification at the upper level (outside the function).

We now have a list of problem patients and can examine the origin of these problems at our leisure, although this is not the purpose of this document:

```
> pat.list

 [1] "1"   "10"  "11"  "12"  "13"  "14"  "15"  "16"  "17"  "18"  "18_" "19"
[13] "2"   "20"  "3"   "4"   "6"   "7"   "8"   "9"

> Err

[1] "11"  "12"  "13"  "14"  "15"  "18_" "6"   "8"   "9"
```

# 3 Efficiency

## 3.1 Why not switching to Rdcm format?

Espadon has its own file storage system in Rdcm format, which is much more efficient in terms of reading speed and disk space than the DICOM format. If you have room on your working disk, it may be worth wasting some time and space to convert DICOMs to Rdcm format once and for all.

The following code will clone the directory containing the patient database and convert their exams to Rdcm format:

```
pat.src.db <- "E:/db"
pat.list <- dir (pat.src.db)

pat.dst.db <- "E:/db_Rdcm"
dir.create (pat.dst.db)

for (pat in pat.list) {
  print(pat)
  dicom.to.Rdcm.converter (file.path (pat.src.db, pat),
                           file.path (pat.dst.db, pat),
                           verbose = FALSE)
}
```

Now we have a directory "`E:/db_Rdcm`" on which we can work safely and with an appreciable saving of time for most operations.

In this example, the original database in DICOM format needs 4.04 GB of storage space. Once converted into Rdcm, it take 1.34 GB. Despite this reduction, no data is lost as Rdcm contains all the DICOM information. Moreover, in DICOM format, we had 7209 files, in RDCM, only 205 (one by exam).

In terms of timing, the conversion (for 20 patients and 205 exams) took 7.6 minutes, but keep in mind you have to do it once.

Now, let us see loading time. The only difference between DICOM and Rdcm format is the loading instruction, repectively `load.patient.from.dicom` and `load.patient.from.Rdcm`:

```
pat.db <- "E:/db_Rdcm"
pat.list <- dir (pat.db)
now <- Sys.time()
pdf ("pats_obj.pdf")
for (p in pat.list) {
  print (p)
  pat <- load.patient.from.Rdcm (file.path (pat.db, p))
  D <- load.obj.data (pat$rtdose[[1]])
  CT <- load.obj.data (pat$ct[[1]])
  S <- load.obj.data (pat$rtstruct[[1]])
  display.obj.links (pat)
}
dev.off ()
Sys.time() - now
```

For the twenty patients (metrics depend on the size of your specific exams):

| | From DICOM | From Rdcm |
|---|---|---|
| Loading the patient-oriented view only | 5.9 min | 0.5 s |
| Loading one rt-dose + CT + rt-struct | 10.25 min | 56 s |

Make your choice…

## 3.2 Why not parallelizing data processing?

Another way to save time is to use parallel computing. Regarding parallel computing, one can think of using graphical accelerators. This is a complex subject that is beyond the scope of this document (apart from the fact that parallel computing can be complicated in R). However, one can easily use several CPU-cores and assign a patient to each core. This is precisely what we will explore here.

Again, the developers of R make it easy for us by providing simple and effective mechanisms to do the job using `parallel`, `doParallel` and `foreach` packages, now included in R-base. Simple, but sometimes, subtle…

In practice, when you have several CPU cores, R will sequentially distribute the work to each core. Once all the cores have been used, it will wait until all the operations have been completed before starting the process again, in order to synchronize the results, until the data to be processed is exhausted. Therefore, it is not possible to place graphs in a pdf file, as we have done so far. Indeed, these graphs are displayed when they are calculated, in an order that depends on the complexity of the calculation and not on the patient index. In

practice, this is of no consequence since parallel computing is essentially used to calculate and all operations can always be displayed once they have been completed.

It is also important to bear in mind :

i) that disk access cannot be parallelized, which inevitably slows down the process,
ii) that the memory is shared by all the cores, so beware of overloading and
iii) it is difficult to stop your code, once launched you must wait the completion.

All this being integrated, it's still nice to see your CPU finally working at full capacity until it needs its fan to cool down. And above all, to have the result of your computations in about 4 to 8 times less time than usual (depending on your computer, disk access, memory…).

Considering what we said above, we have to change `my.skin.dose` function. It will not display the results anymore, but rather store them in a list. This list can also contain computation results, for instance:

```
my.skin.dose <- function (pat.dir) {
  pat <- load.patient.from.Rdcm(pat.dir)

  S <- load.obj.data (pat$rtstruct[[1]])
  D <- load.obj.data (pat$rtdose[[1]])

  B.pat <- bin.from.roi (D, S, roi.name = "body")
  B.pat.neg <- bin.inversion (B.pat)
  B.pat.neg.expansion <- bin.dilation (B.pat.neg, radius = 3)
  B.skin <- bin.intersection (B.pat, B.pat.neg.expansion)

  D.skin <- vol.from.bin (D, B.skin)
  H <- histo.vol (D.skin, breaks = seq (0, 60, by=0.1))
  DVH <- histo.DVH (H)

  M.pat <- mesh.from.bin (B.pat)

  display.3D.mesh (M.pat, col="burlywood")
  f <- tempfile("dum", fileext=".png")
  rgl.snapshot (f)
  rgl.close ()
  png <- readPNG(f)

  return (list (B.skin = B.skin, DVH = DVH, png = png))
}
```

Note that we have stored the image in a temporary file to avoid access conflicts if, for example, two processes were trying to access the same file.…

Then, let us install the packages and see how many cores we have at hand:

```
require (parallel)
require (doParallel)
require (foreach)

detectCores (logical = FALSE)

[1] 8
```

We have eight CPU-cores. We decide to use seven (in order to be able to continue this document while the computer is calculating). The parallel code is the following:

```
pat.db <- "E:/db_Rdcm"
```

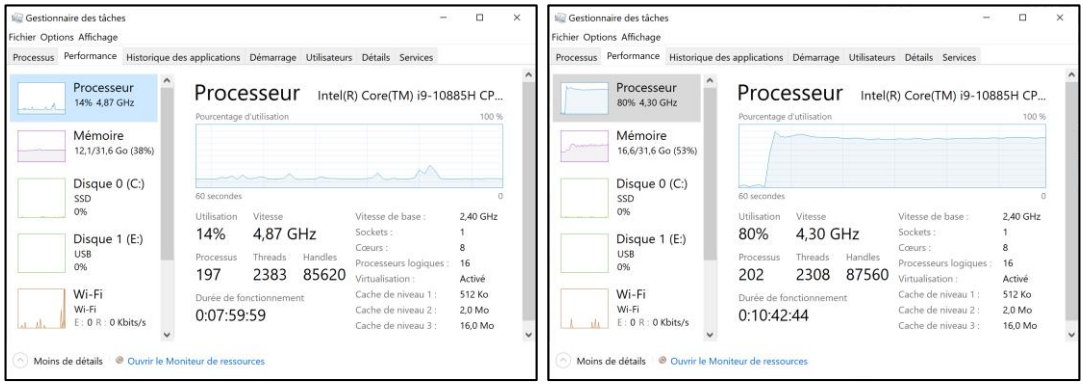
```
pat.list <- dir (pat.db)

cl <- makeCluster (7)
registerDoParallel (cl)

results <- foreach (p = pat.list) %dopar% {
  require (espadon)
  require (rgl)
  require (png)
  res <- tryCatch (my.skin.dose(file.path (pat.db, p)),
           error = function (e) {})
  if (is.null (res)) res <- NA
  return (res)
}

stopCluster (cl)
```

It need some explanations…

1) `makeCluster` and `registerDoParallel` are used to prepare the CPU cores to receive the code. Here, we will use seven cores.
2) `foreach (p = pat.list)` will execute the code in parenthesis as we did before, using `for (p in pat.list)`, but this time in parallel. Note the difference in syntax.
3) %dopar% means the code will be executed in parallel. You can try %do% if you want to execute it as usual.
4) In fact, each time the code is executed, a new instance of R is created on the CPU. This instance does not know the environment, so you will have to wake-up the packages you need in this section. Here, we need `espadon`, `rgl` and `png`.
5) As usually, the code generates errors we have to handle. Note here, the `error` process does nothing. All results are returned in `res`. If res is `NULL`, we replace it by `NA` and return the result. `foreach` will gather the results in a list.
6) Last, the clusters are released.

Running this code, you would get a list (stored in `results`) whose elements contain the execution results for each patient (eventually `NA` if the computation went wrong).

CPU usage can be seen bellow. On the left, in standard mode, only one CPU is used. On the right, in parallel mode, seven CPU are computing:




CPU charge when running code in standard mode (on the left) and in parallel (on, the right). Running your code in parallel on N cores does not necessary mean you will gain N in time. Gain also depends on file transfer (not parallelizable) and cores synchronization.

Depending on your problem and its limitations (loading or calculation), you will see a time saving ranging from a factor of 1 (bad luck) to 7 (in this case).

Parallel computing is a complex and constantly evolving subject. We have presented only its simplest expression here and we encourage you to explore it in more detail if you feel the interest or the need.

# 4 What we learned

The massive use of medical data in DICOM format is rather quickly cumbersome in terms of loading time and/or calculations.

We have seen, through a simple example, how to develop the code on a particular patient and generalise it to a multitude of patients by encapsulating it in a function. It goes without saying that programming is very often based on a priori assumptions that are detrimental to the robustness of the code. These assumptions are usually the source of errors and are corrected gradually as the code is written. However, to move forward, it is useful to run the code knowing that it may encounter difficulties. R's `tryCatch` mechanism simplifies the handling of these difficulties by allowing execution to continue, even in the event of errors. It is then up to the developer to understand their origin and correct them.

In all cases, loading the data is a time-consuming process, especially when the patient has had many examinations. We have seen that converting from DICOM to Rdcm format is a simple and effective way of greatly reducing these incompressible transfer times.

Another time-saving approach is to parallelize the code. This is usually a complex operation, however, if we just parallelize on the patients, R offers simple and efficient mechanisms to access this kind of acceleration.

Nevertheless, computing on 3D data is necessarily greedy in terms of memory usage. A patient quickly occupies several GB of RAM and if you want to be quiet (especially if you plan to parallelize your work), you will need a machine with a lot of RAM anyway; 8 GB is a strict minimum, consider 16, even 32 GB of memory...