ESAIM: PROCEEDINGS AND SURVEYS, **September 2014, Vol. 45, p. 239-246**
J.-S. Dhersin, Editor

USING LEPTON FOR DOCUMENTING SOURCE CODE : A GUIDED EXAMPLE IN COMPUTER VISION

Sébastien Li-Thiao-Té[1]

Abstract. This paper describes a method for analyzing the structure and documenting a fairly large chunk of source code for image analysis with a literate programming tool called Lepton. We propose a step-by-step approach to deconstructing the source code and indicate where Lepton's specific features are most useful.

Résumé. Ce manuscrit présente le logiciel Lepton et comment l'utiliser pour analyser la structure et documenter le code source d'un programme d'analyse d'images. Nous proposons un guide étape par étape et présentons les fonctionnalités du logiciel Lepton là où elles sont particulièrement adaptées aux différentes opérations.

Introduction

Computers have become unavoidable assistants to modern science. Computers measure and store data sets, analyze data sets, perform large-scale simulations, and are also necessary for communication and publication of the scientific results. However, there is growing realization that our current procedures for collaboration and publication are not optimal. In many fields of study such as econometrics [2], biostatistics [7] and image analysis [6], many published scientific results cannot be reproduced independently because essential elements of the methods are not available or insufficiently documented.

In this manuscript, we propose a method for documenting computational research results that facilitates re-use by collaborators or readers as well as re-appropriation, that is to say in-depth understanding of the contents of the research. This method is supported by Lepton [3], a freely available software tool developped in house for this specific purpose. It is well suited for doing research but also for adding documentation to existing methods. This tool has been described previously in [4] from the point of view of reproducible research and in [5] for its use in teaching mathematics (random exercises with solutions and project reports). Here we start from a rather large piece of image analysis code without a manual and illustrate how Lepton can be used to understand how the source code works and how to use the corresponding program.

1. Documentation: objectives and requirements

1.1. Investing in documentation

In our opinion, one major rationale for writing documentation is to convey knowledge to our future self. This is what turns random bits of digital information into usable and meaningful media. Proper documentation should be written for the future so that the work remains live and usable long after its production. In that sense, documenting is a long-term investment, and we have its returns when we want to reuse and modify it.

[1] Laboratoire LAGA, CNRS (UMR 7539), Université Paris 13, France e-mail: lithiao@math.univ-paris13.fr

For instance, it is very common to redraw figures or rerun analyses of datasets to answer reviewers' comments or match the editing policy of a journal. Sometimes the figures must simply be redrawn without color, sometimes larger simulations or benchmarks are called for. When presenting your work at a conference or comparing with newer work, figures may need to be adapted for the presentation slides or for the conference proceedings. Being able to re-generate figures or other results is very helpful in these situations, especially when it is easy to find what to modify.

Reviewing past work is another reason for thorough documentation. For example, a system that automatically records the commands and parameters used for generating figures or lengthy simulations is a must-have when discussing the results with collaborators. A framework that is simple from a human perspective also allows collaborators to experiment your method with different choices of parameters, input data, and compare with other work.

Additionaly, we believe that writing documentation is a way to step back, and look at the work from another angle. This improves the quality of the work by promoting elegant rather than quick and dirty solutions and source code. We may find more flexible or generic implementations, making it easier to re-use the work later or to adapt it to a different context.

Whatever the reason, it should be convenient to reach these objectives.

1.2. **Proposed method**

Our proposed documentation framework hinges on three compoments:

- a package format that holds all the elements in the research project,
- a software tool for extracting the elements from the package,
- a documentation format that is compatible with existing practices and automatically provides information on the contents of the package.

Lepton uses a file format inspired by literate programming [1]. In this approach, authors are encouraged to focus on writing documentation in the format of their choice. This manuscript is in fact a Lepton file written with LaTeX as the documentation format. All other elements such as source code, scripts, input data and results are embedded inside the documentation in blocks (called *code chunks*) of the form :

Code chunk 1: ≪`lepton`≫

```
<<name options>>=
contents
@
```

The Lepton software is responsible for extracting the contents of code chunks, and producing a bona fide documentation file that can be processed by LaTeX. During this process, several operations can be applied to the contents of code chunks, depending on the options specified by the author:

- `-write` writes the contents of the code chunk to disk (chunk name is used for the filename)
- `-exec interpreter` sends the contents to an external process for execution (Unix shell, Matlab, Scilab, etc.)
- `-chunk format -output format` controls how the chunk contents and outputs are embedded in the resulting documentation
- `<<chunk_ref>>` is an abbreviation for the contents of another code chunk.

In the literate programming paradigm, authors can use the above features to manipulate source code, regardless of the constraints of the programming language. For instance:

- source code can be divided into meaningful chunks
- code chunks can appear anywhere, and especially below the corresponding algorithmic description or a specification to complement the documentation,
- test cases can be defined and executed next to the code

- benchmarks defined inside the documentation will have their results automatically embedded inside the documentation and updated upon source code modification.

Note that providing the source code that corresponds to an algorithmic description does not require additional work, but allows the reader to check if the two are coherent.

We propose to use literate programming with Lepton in incremental stages. First, we package all the necessary elements into a single document and set up extraction properly. Second, we make sure that source code compiles. Then, we make sure that the binaries can be executed and set up some tests. Finally, we analyse and document the functions in the source code and the methods. The documentation process may be interrupted at any stage, Each stage provides a given level of usability and reproducibility.

1.3. **Image analysis example**

As an example, we relate our experience when trying to use and understand an image analysis library developped in house without Lepton. This source code comes as a monolithic C source file of 5069 lines with comments, and implements five image compression methods. There is no installation or usage manual.

2. First stage : make a documented package

Start with a working installation of Lepton. Lepton is distributed as a standalone executable for Linux, and downloading the binary file from `http://www.math.univ-paris13.fr/~lithiao/ResearchLepton/Lepton.html` should be sufficient. On 64bit systems, the executable runs as a 32bit program and thus requires the basic 32bit libraries. The usage manual and several examples are also available on the website.

For this example, we open a new file named `readme.nw` in a text editor like Emacs or Vim. This file will act both as a LaTeX documentation, and a package that will contain our comments as well as the source code. At this stage `readme.nw` is pure documentation. Running Lepton on this file produces a `readme.tex` file that is identical, and we verify that it can be processed by LaTeX.

Then we package each file in the project by inserting its contents in a code chunk, and describe it in the LaTeX documentation. This code chunk should have option `-write` and the chunk name should be the name of the file Lepton will create. In our example, there is only one C source file :

Code chunk 2: ≪`readme.nw`≫
```
% LaTeX documentation
This project implements all five methods in a single C source file below.
<<bigfile.c -write>>=
C source code
@
```

Input files and other files can be added to `readme.nw` in the same way.Again, we run Lepton, and check that `bigfile.c` is correctly re-created by Lepton.

At this stage, you should have a package containing all the necessary files and their description in a single Lepton file. Lepton is able to extract those elements from the source, and documentation can be compiled. If you decide to stop working at this stage, `readme.nw` is a package containing all the files related to the project and their description.

Note that when sending `readme.nw`, collaborators do not require Lepton for accessing the contents of the package. The file `readme.nw` can be opened and modified in any text editor, and the files can be extracted by copy-and-paste. We suggest that only files of moderate size be included in `readme.nw` in order to preserve the readability of both the documentation and the LaTeX source. Authors should only include large files as a temporary measure before decomposition into meaningful elements. Lepton provides the `\Linput` directive for assembling large projects.

3. Second stage : compile

Before attempting to compile the source code, we suggest including the machine configuration in the `readme.nw` file, either by capturing the output of a `configure` script, or by running system information commands. This provides information that can be used for debugging when compilation fails or used to indicate a working configuration upon success. To do this, we use a code chunk containing the following commands. With the option `-exec shell`, Lepton starts a Unix shell, executes the commands, and outputs the results.

Code chunk 3: ≪`configuration`≫

```
uname -a
COLUMNS=115 dpkg -l gcc libc6 | tail -n 3
```

Interpret with `shell`

```
Linux lepton 3.12-1-686-pae #1 SMP Debian 3.12.6-1 (2013-12-21) i686 GNU/Linux
+++-=====================-===============-===============-==================================================
ii  gcc                   4:4.8.1-1       i386            GNU C compiler
ii  libc6:i386            2.17-97         i386            Embedded GNU C Library: Shared libraries
```

To compile the source code, we use the same mechanism. We write the compilation commands in a code chunk with the option `-exec shell`. As the program uses the math library, we need to add the `-lm` flag. When gcc reports no error, it produces no output.

Code chunk 4: ≪`shell`≫

```
gcc -lm bigfile.c -o program.bin
```

Interpret with `shell`

For larger programs or more complex scenarii, it is often recommended to use a makefile. This makefile can be embedded into `readme.nw` in its own code chunk, and executed by a shell command. The Unix shell can also be used to run documentation tools such as dependency graph generators, static analysis code tools, etc. An advantage of using Lepton, is that the outputs are automatically collected and that the produced documentation is always in a coherent state.

At this stage, you should have a self-compiling program. If you send this, the recipient can extract the files, compile the program, but there is still no documentation about how the method works.

4. Third stage : execute

The compiled program is now ready for testing. Although you could start a terminal and type the commands, we suggest that you again use a code chunk executed by the shell in order to capture the outputs. Our program does not read command-line parameters, but asks the user for manual input. Consequently, we have to redirect the parameters to the standard input channel.

Without a manual or a test case, we had to test blindly the program on a random image, and see the results. When that failed, we had to go back to the author of the program to inquire the meaning of the different parameters, and how to retrieve the solution. Usage information is now contained in the following code chunk, and the results are displayed by LaTeX. The code chunk also contains commands for converting the image format used by the program into PNG for inclusion in the LaTeX documentation.

Code chunk 5: ≪execute≫

```
echo "1 peppers.pgm 0 9 3 sol.pgm dec.pgm" | ./program.bin
convert peppers.pgm peppers.png
convert sol.pgm sol.png
convert dec.pgm dec.png
```

Interpret with `shell`

```
            Application for compression/approximation : choose the method

 Linear     Tensor Product Approach  (1)

 ENO        Tensor Product Approach  (2)

 ENO-SR     Tensor Product Approach  (3)

 ENO    NON Tensor Product Approach  (4)

 ENO-SR NON Tensor Product Approach  (5)

Original Image ?Threshold = Finer level = Lower level = Type of normalization L1 (1), L2 (2) or Linf (3)Solution ?Decomposition  ?lectu
Compression ratio 1.000000
Error in L2 norm 0.000026
PSNR :=139.972992
Error in L1 norm 0.009016
Error in L_inf norm 0.000214
```

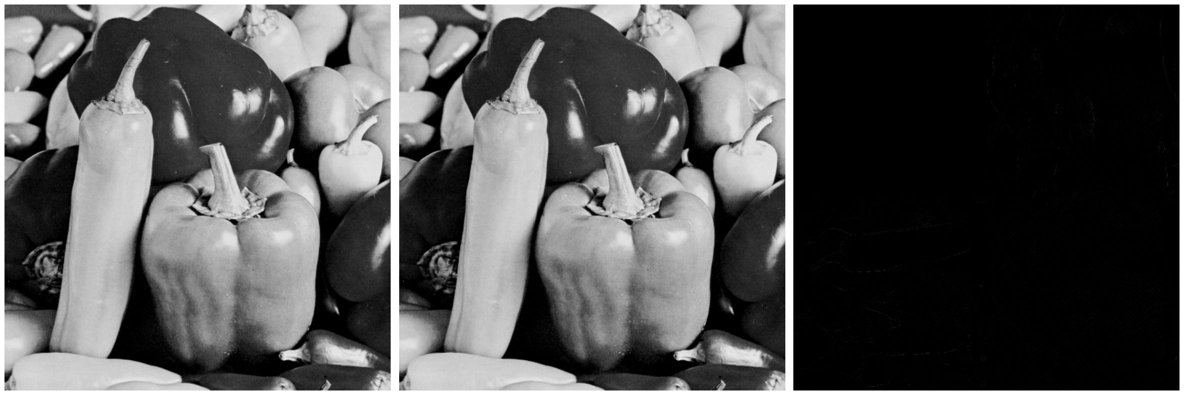

FIGURE 1. Input image (left), reconstruction (middle). The source code performs a kind of wavelet decomposition. The image on the right represents the decomposition coefficients.

At this stage, you should have an executable program. In this case, running Lepton on `readme.nw` will apply the program to `peppers.pgm`. The results are automatically included and updated in the produced documentation and are also available as image files in the current directory. The input image can be replaced for testing other inputs, but the filename must be the same.

5. Fourth stage : analyse

We can now begin to analyse the source code itself and make modifications to the program. Our first goal was to generate benchmarks of the image analysis routines in order to evaluate their performance on a series of images. This entails converting the current parameter passing mechanism to using command line parameters. We also separated the different methods and create one program for each method.

When looking at the source code, we noticed that the main function is just a wrapper to several functions, and that each function performs parameter input separately.

Code chunk 6: ≪main≫

```
/*..................................................................*/
main(){
  int ch;
printf("\n           Application for compression/approximation : choose the method       \n ");
printf("\n Linear     Tensor Product Approach  (1)\n ");
printf("\n ENO        Tensor Product Approach  (2)\n ");
printf("\n ENO-SR     Tensor Product Approach  (3)\n ");
printf("\n ENO    NON Tensor Product Approach  (4)\n ");
printf("\n ENO-SR NON Tensor Product Approach  (5)\n ");
printf("\n");

scanf("%d", &ch);

      if(ch==1) linear();
 else if(ch==2) enoTP();
 else if(ch==3) enosrTP();
 else if(ch==4) enoNTP();
 else if(ch==5) enosrNTP();
}
```

Our first move was to write five main functions from the same template in order to obtain one binary executable for each function. For instance, this is the program for the `linear` method.

Code chunk 7: ≪linear.c≫

read_parameters

```
int main(int argc, char** argv)
{
  <<read_parameters>>
  linear(init,fin,inter,p,e0,resn);
}
```

All five programs now use the same procedure for parsing the command-line parameters. This is defined once in code chunk 8 and shared by using a chunk reference.

Code chunk 8: ≪read_parameters≫

```
if (argc != 6+1) {
  printf("This program expects 6 argument\n");
  printf("Usage : linear original.pgm solution.pgm decomposition.pgm threshold flevel llevel type\n");
  printf("        original.pgm : source image, square, power of two, PGM format\n");
  printf("        solution.pgm : source image, square, power of two, PGM format\n");
  printf("        decomposition.pgm : source image, square, power of two, PGM format\n");
  printf("        threshold : floating-point number\n");
  printf("        llevel : int, lower level\n");
  printf("        type : int, type of normalization L1 (1), L2 (2) or Linf (3)\n");
  return EXIT_FAILURE;
}

char* init = argv[1];    /* source image */
char* fin = argv[2];     /* output image */
char* inter = argv[3];   /* decomposition image */
float p = atof(argv[4]); /* Threshold */
int e0 = atoi(argv[5]);  /* lower level */
int resn = atoi(argv[6]);/* type of normalization */
```

Similarly, the logic for reading the input image and writing the results to disk is shared between the five programs. At this point, we discovered that the methods could only be applied to square images of size power of two. This was corrected by copying the image contents into the smallest square image. By using a chunk reference, we ensure that this modification applies to all five programs correctly.

At this stage, `readme.nw` contains implementation details with embedded source code. Code is divided in small meaningful chunks that are organized according to their meaning rather than in the order dictated by the compiler. Shared code is re-used by using chunk references. The program can now be applied on multiple images with a (hidden) shell script, output is set for direct inclusion into the LaTeX documentation in Figure 2.

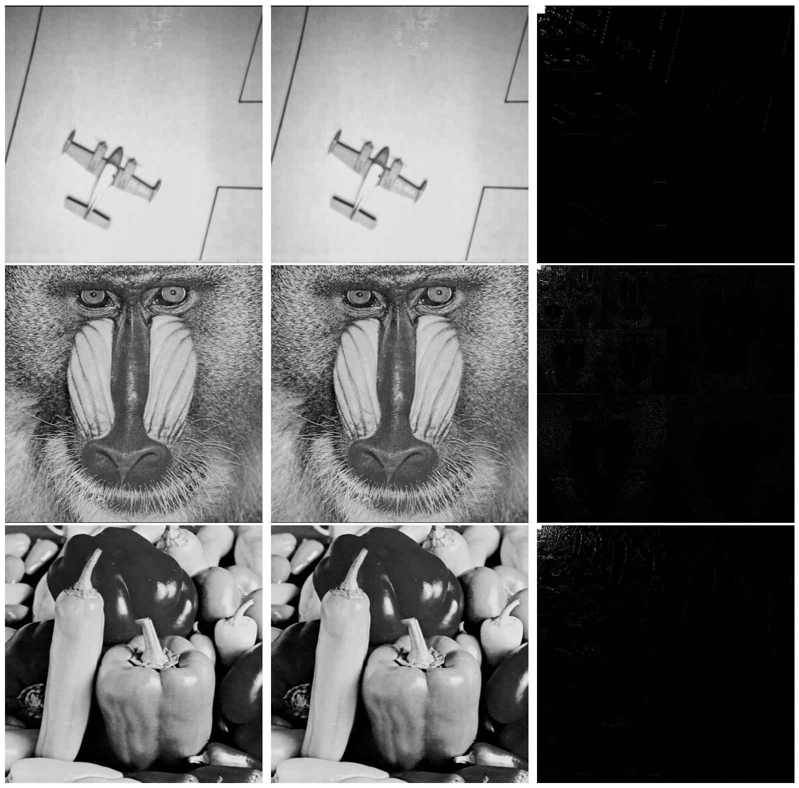

FIGURE 2. Input image (left), reconstruction (middle), representation of the coefficients (right)

6. Next steps and conclusion

We have progressively modified and documented a fairly large C source file containing several image analysis programs. With the help of Lepton, we have first packaged and documented the project files, then added instructions for compilation and usage. Finally, we modify the source code in order to run a benchmark of the program.

Each stage provides increasing levels of reproducibility and re-usability of the program itself and also of our documentation work. Packaging makes the whole project available for sharing with collaborators. Compiling and execution instructions provide the means to use the method. Implementation details provide information for modifying the method, and often improves the quality, modularity and correctness of source code.

References

[1] Donald E. Knuth. Literate programming. *THE COMPUTER JOURNAL*, 27:97–111, 1984.

[2] Roger Koenker and Achim Zeileis. On reproducible econometric research. *Journal of Applied Econometrics*, 24(5):833–847, 2009.

[3] S. Li-Thiao-Té. Lepton user manual. `http://www.math.univ-paris13.fr/~lithiao/ResearchLepton/Lepton.html`.

[4] Sébastien Li-Thiao-Té. Literate program execution for reproducible research and executable papers. *Procedia Computer Science*, 9(0):439 – 448, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[5] Sébastien Li-Thiao-Té. Literate program execution for teaching computational science. *Procedia Computer Science*, 9(0):1723 – 1732, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[6] Nicolas Limare and Jean-Michel Morel. The ipol initiative: Publishing and testing algorithms on line for reproducible research in image processing. *Procedia Computer Science*, 4(0):716 – 725, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[7] A. Rossini and F. Leisch. Literate statistical practice. *UW Biostatistics Working Paper Series*, page 194, 2003.